

**CASE STUDY REPORT**

**DESIGN AND ANALYSIS OF ALGORITHMS**

**19CSE302**

**TOPIC: TEXT COPRESSION USING HUFFMAN CODING**

**BY:**

|  |  |
| --- | --- |
| **HEMANTH** **BYSANI** | **BL.EN.U4CSE20057** |
| **INDIRA** **KUMAR** **A K** | **BL.EN.U4CSE20060** |
| **AASHRITHA DANDA** | **BL.EN.U4CSE20004** |

# PROBLEM STATEMENT

* To design a text compressor using Greedy approach known as Huffman coding.
* There are many situations in which fast data transmission is essential, such as when real-patient data or military coordinates, our goal is to meet is need for speed by developing solutions that facilitate faster data transfer over the internet or any other medium.

# INTRODUCTION

* The exponential growth of data is a significant challenge. Currently, it is estimated that at least 2.5 quintillion bytes of data are produced daily, and this data is typically transferred using ASCII code over the internet. To improve the speed of data transmission, our goal is to minimize the size of this vast amount of data as much as possible.
* The methodology that we have chosen to solve the aforementioned problem statement is Huffman coding.
* Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.
* Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

# ABSTRACT

* The time required to encode each unique character based on its frequency has a complexity of O (n logn).
* The approach used here is Greedy Method and the algorithm is Huffman coding.
* The complexity of extracting the minimum frequency from the priority queue 2\*(n-1) times is O(log n). When combined with the time required for encoding unique characters based on their frequency, the overall complexity is O(n logn).
* Huffman coding is a widely used method for data compression and is implemented in various compression formats such as TAR, GZIP, BZIP2, and PKZIP.
* We can implement a multi layered encoding as well by re-encoding the encoded binary values to higher order.
* Huffman coding is a lossless data compression algorithm.
* We have used dictionary to store the encodings of each character and its binary value from Huffman tree.

# ALGORITHM

An algorithm for Huffman encoding and decoding:

## Huffman encoding:

* Create a frequency table for the characters in the data, with the characters as keys and their frequencies as values.
* Create a heap of nodes, with each node representing a character and its frequency.
* While the heap has more than one node, do the following:

a. Pop the two nodes with the lowest frequencies from the heap.

b. Create a new node with a frequency equal to the sum of the frequencies of the popped nodes. c. Set the left child of the new node to the node with the lower frequency, and the right child to the node with the higher frequency.

d. Add the new node to the heap.

* The remaining node in the heap is the root node of the Huffman tree.
* Traverse the Huffman tree and assign codes to each character, with a "0" for a left child and a "1" for a right child.
* Use the assigned codes to compress the data.

## Huffman decoding:

* Create a decoding tree using the same Huffman tree used for encoding.
* Initialize a current\_node variable to the root of the decoding tree.
* For each bit in the encoded data, do the following: a. Set the current\_node to its left child if the bit is a "0", or to its right child if the bit is a "1". b. If the current\_node is a leaf, add the character it represents to the decoded data and set the current\_node back to the root of the decoding tree.
* Return the decoded data.

# IMPLEMENTATION

CODE:

import heapq

import collections

# ENCODING

def huffman\_encoding():

    # Read the input file and count the frequency of each character

    with open("input.txt", "r") as file\_input:

        freq = collections.Counter(file\_input.read())

    # Build the Huffman tree

    heap = [[weight, [symbol, ""]] for symbol, weight in freq.items()]

    heapq.heapify(heap)

    while len(heap) > 1:

        lo = heapq.heappop(heap)

        hi = heapq.heappop(heap)

        for pair in lo[1:]:

            pair[1] = '0' + pair[1]

        for pair in hi[1:]:

            pair[1] = '1' + pair[1]

        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    # Generate the encoding for each character

    global encoding

    encoding = {}

    for pair in heapq.heappop(heap)[1:]:

        encoding[pair[0]] = pair[1]

    global before\_compression

    with open("input.txt", "r") as file\_input:

        data = file\_input.read()

        before\_compression = len(data) \* 8

    # Write the encoded output to the output file

    # print(encoding)

    with open("encoding.txt", "w") as file\_outt:

        file\_outt.write(str(encoding))

    with open("encoded.txt", "w") as file\_output:

        with open("input.txt", "r") as file\_input:

            # print(file\_input.read())

            file\_output.write("".join([encoding[ch] for ch in file\_input.read()]))

huffman\_encoding()

# GAIN CALCULATION

after\_compression = 0

symbols = encoding.keys()

for symbol in symbols:

    with open("input.txt", "r") as file\_input:

         data = file\_input.read()

    count = data.count(symbol)

    after\_compression += count \* len(encoding[symbol])

print("Space usage before compression (in bits):", before\_compression)

print("Space usage after compression (in bits):",  after\_compression)

# DECODING

def huffman\_decoding():

    # Read the Huffman encoding from a file

    encoding = {}

    try:

        with open("encoding.txt", "r") as file\_input:

            for line in file\_input:

                for elements in line.split(","):

                    # print(elements)

                    ch, code = elements.replace("'","").replace("{", "").replace("}","").strip().split(": ")

                    encoding[ch] = code

            # print(encoding)

    except FileNotFoundError:

        print("Error: The file 'encoding.txt' does not exist.")

        return

    # Read the encoded data from a file

    try:

        with open("encoded.txt", "r") as file\_input:

            encoded = file\_input.read()

    except FileNotFoundError:

        print("Error: The file 'encoded.txt' does not exist.")

        return

    # Decode the encoded data

    decoded = ""

    current\_code = ""

    if encoded:

        for ch in encoded:

            current\_code += ch

            if current\_code in encoding.values():

                # print(current\_code)

                value = {i for i in encoding if encoding[i]==current\_code}

                decoded += (list(value)[0])

                current\_code = ""

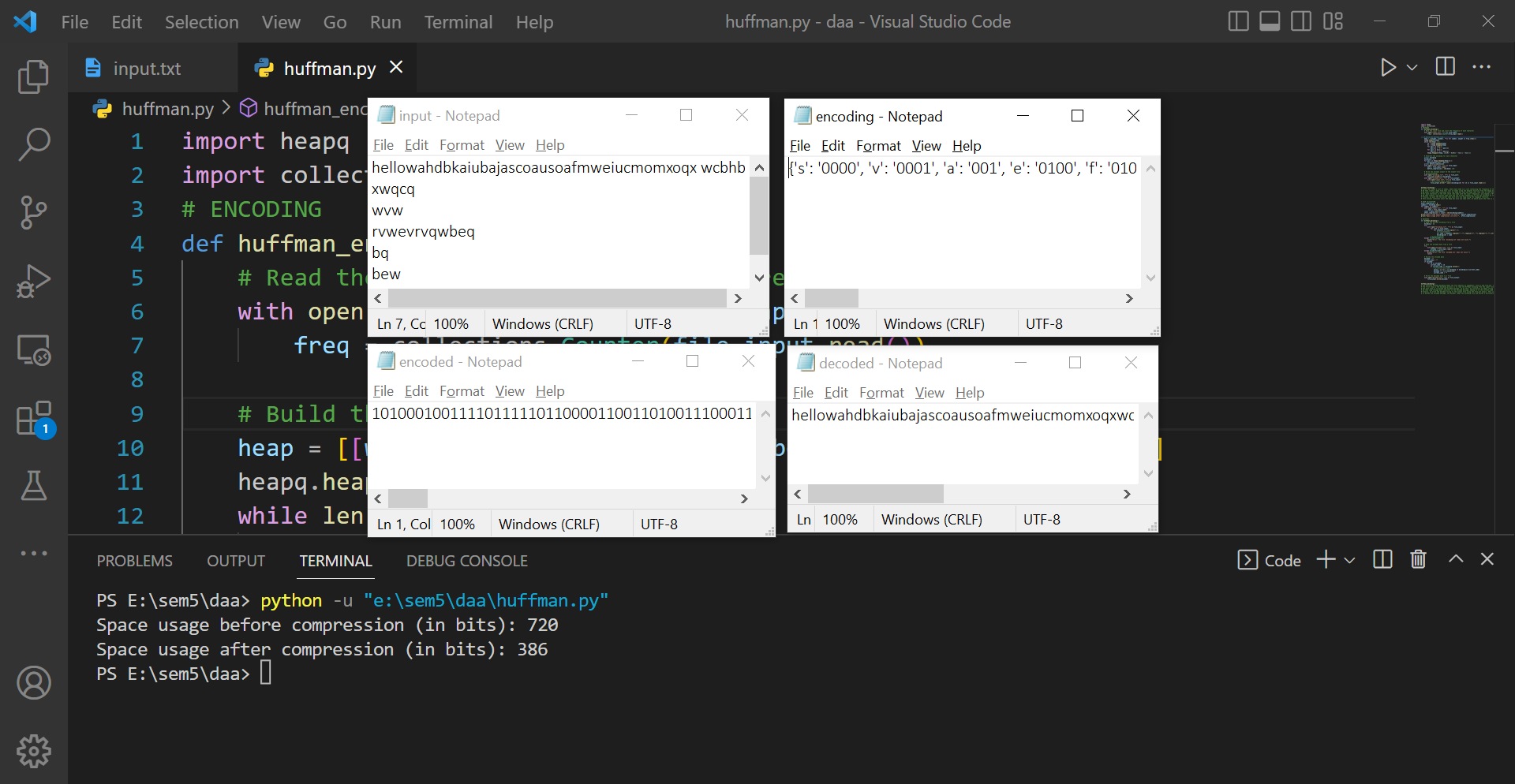
    # Write the decoded data to a file

    with open("decoded.txt", "w") as file\_output:

        file\_output.write(decoded)

huffman\_decoding()

OUTPUT:



# EXPLANATION

In our implementation of the Text compression, we used the Huffman coding algorithm. Huffman coding is a data compression algorithm that assigns variable-length codes to characters in a way that minimizes the overall number of bits used to represent the characters. The algorithm works by creating a binary tree based on the frequencies of the characters in the data, with the most frequent characters at the root of the tree and the least frequent characters at the leaves. Each character is then assigned a code based on the path from the root to the leaf containing that character, with a "0" representing a left child and a "1" representing a right child.

Huffman coding is widely used in lossless data compression applications, such as the compression of text and image files. It is known for its simplicity and efficiency, and is used in many standard compression formats, such as GZIP and PKZIP.

In encoding:

* First, we create a list of nodes, where each node is a list containing the frequency of the character and a list of the character and its code. For example, the node for the character "a" with frequency 3 would be [3, ["a", ""]].
* We then create a heap from this list of nodes using the heapify function from the heapq module. This function rearranges the list in-place to form a heap.
* We enter a loop that continues until the heap has only one node left (the root of the tree). Within the loop, we pop the two nodes with the lowest frequencies off the heap using the heappop function. These nodes will become the children of a new parent node.
* For each character in the left child (the node with the lower frequency), we prepend a "0" to its code. For each character in the right child (the node with the higher frequency), we prepend a "1" to its code.
* # We then create a new parent node with the sum of the frequencies of the two children as its frequency, and the two children as its children.
* # Finally, we push the new parent node back onto the heap using the heappush function.
* # This process continues until the heap has only one node left, at which point the tree is complete. The codes for each character can then be extracted from the tree by traversing the tree and looking at the code for each character.

In decoding:

* The function huffman\_decoding takes two file objects as arguments: file\_in and file\_out. The file\_in object is used to read the Huffman encoding and the encoded message from the input file, and the file\_out object is used to write the decoded message to the output file.
* The first step in the decoding process is to read the Huffman encoding from the input file. This is done by looping through each line in the file and splitting it on the colon character (":"). The left side of the colon is the character, and the right side is the code. These values are then added to the encoding dictionary, with the code as the key and the character as the value.
* The next step is to decode the encoded message. To do this, we initialize an empty string called decoded and a string called current\_code that will be used to store the code that we are currently building up as we read the encoded message.
* We then loop through each character in the encoded message, adding it to the current\_code string. If the current\_code string is in the encoding dictionary, we append the corresponding character to the decoded string and reset the current\_code string to be empty. If the current\_code string is not in the encoding dictionary, we continue adding characters to it until we find a matching code.
* Finally, we write the decoded string to the output file using the file\_out object. This completes the decoding process.
* To decode the encoded message, the decoder reads the encoding file and builds a dictionary that maps the codes to the corresponding characters. It then reads the encoded message one character at a time, adding the characters to a current code until it finds a matching code in the dictionary. When a matching code is found, the decoder adds the corresponding character to the decoded message and resets the current code to be empty. This process is repeated until the entire encoded message has been decoded.
* We can calculate the gain from encoding by implementing a simple function where we calculate the number of bytes before compression and calculate the number of bytes after compression using the encoding dictionary and the input.txt file, it can be seen from our results that we obtain a 2x lossless compression while using this encoding algorithm.

# BIBLIOGRAPHY

[*https://www.gatevidyalay.com/huffman-coding-huffman-encoding/*](https://www.gatevidyalay.com/huffman-coding-huffman-encoding/)

> Gate Vidyalay Huffman Encoding

[*Data-Structures-and-Algorithms-with-Python/Huffman\_Encoding at main · YCAyca/Data-Structures-and-Algorithms-with-Python (github.com)*](https://github.com/YCAyca/Data-Structures-and-Algorithms-with-Python/tree/main/Huffman_Encoding)

>GitHub implementation article

[*https://www.programiz.com/dsa/huffman-coding#:~:text=Huffman%20coding%20first%20creates%20a,concept%20of%20prefix%20code%20ie.*](https://www.programiz.com/dsa/huffman-coding#:~:text=Huffman%20coding%20first%20creates%20a,concept%20of%20prefix%20code%20ie)

> Programiz Huffman Coding

[*https://www.youtube.com/watch?v=co4\_ahEDCho*](https://www.youtube.com/watch?v=co4_ahEDCho)

> Abdul Bari Huffman Coding