#Write a program to implement Huffman Encoding using a greedy strategy.

import heapq

class Node:

def \_\_init\_\_(self, freq, symbol, left=None, right=None):

self.freq = freq

self.symbol = symbol

self.left = left

self.right = right

self.huff = ''

def \_\_lt\_\_(self, nxt):

return self.freq < nxt.freq

def print\_nodes(node, val=''):

new\_val = val + str(node.huff)

if node.left:

print\_nodes(node.left, new\_val)

if node.right:

print\_nodes(node.right, new\_val)

if not node.left and not node.right:

print(f"{node.symbol} -> {new\_val}")

def build\_huffman\_tree(chars, freq):

nodes = []

for x in range(len(chars)):

heapq.heappush(nodes, Node(freq[x], chars[x]))

while len(nodes) > 1:

left = heapq.heappop(nodes)

right = heapq.heappop(nodes)

left.huff = 1

right.huff = 0

new\_node = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)

heapq.heappush(nodes, new\_node)

return nodes[0]

def get\_user\_input():

chars = input("Enter characters separated by spaces: ").split()

freq = list(map(float, input("Enter frequencies corresponding to the characters, separated by spaces: ").split()))

if len(chars) != len(freq):

raise ValueError("The number of characters must match the number of frequencies.")

return chars, freq

def main():

try:

chars, freq = get\_user\_input()

root = build\_huffman\_tree(chars, freq)

print("Huffman Codes:")

print\_nodes(root)

except ValueError as e:

print(f"Error: {e}")

except Exception as e:

print(f"An unexpected error occurred: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

"""

X1 x2 x3 x4 x5

0.3 0.15 .0.25 0.05 0.25

Qa

### 1. What is Huffman Encoding?

Huffman Encoding is a compression algorithm that assigns variable-length codes to characters in a message based on their frequency of occurrence. Developed by David A. Huffman in 1952, this encoding technique uses shorter codes for more frequent characters and longer codes for less frequent characters, thereby minimizing the total number of bits required to encode a message. This approach is a form of **lossless data compression**, meaning the original data can be perfectly reconstructed from the compressed data without any loss of information.

The algorithm works by creating a binary tree called a **Huffman Tree**. It constructs this tree in a way that assigns binary codes to each character based on its position in the tree. Characters with higher frequencies are closer to the root and thus have shorter codes, while less frequent characters are deeper in the tree, resulting in longer codes.

**Steps in Huffman Encoding**:

1. Calculate the frequency of each character in the message.
2. Construct a priority queue (min-heap) where each character is a node, sorted by frequency.
3. Combine the two lowest-frequency nodes to create a new internal node with a frequency equal to the sum of the two nodes.
4. Repeat the process until only one node remains, which becomes the root of the Huffman Tree.
5. Traverse the tree to assign binary codes to each character, with "0" for left branches and "1" for right branches.

Huffman Encoding is widely used in applications like **file compression** (e.g., ZIP files) and **image compression** (e.g., JPEG format), where efficient data encoding is critical.

### 2. How many bits may be required for encoding the message ‘mississippi’?

Let's calculate the bits required to encode the message "mississippi" using Huffman Encoding.

1. **Character Frequencies in ‘mississippi’**:
   * **m**: 1
   * **i**: 4
   * **s**: 4
   * **p**: 2
2. **Building the Huffman Tree**: Using the characters and their frequencies, we create the Huffman Tree. Each character is assigned a binary code based on its frequency in the tree.
3. **Assigning Codes** (Example of possible codes):
   * Assuming Huffman encoding results in the following (hypothetical) codes:
     + m: 110
     + i: 0
     + s: 10
     + p: 111
4. **Encoding ‘mississippi’**:
   * Converting each character in "mississippi" to its Huffman code: 110 0 10 10 0 10 10 0 111 111 0
   * Total bits = 3 (m) + 1 (i) + 2 (s) + 2 (s) + 1 (i) + 2 (s) + 2 (s) + 1 (i) + 3 (p) + 3 (p) + 1 (i) = 21 bits

Without Huffman encoding, each character could have required up to 4 bits (e.g., ASCII encoding), resulting in 44 bits. With Huffman encoding, we achieve a **compressed size of 21 bits**, effectively reducing storage by more than half.

### 3. Which tree is used in Huffman encoding? Give one Example

In Huffman Encoding, a **binary tree** known as a **Huffman Tree** is used. This tree is a **full binary tree** (every node has 0 or 2 children), and it is constructed based on the frequencies of characters in the input data.

**Example**: Consider encoding the string "abcde" with the following character frequencies:

* **a**: 5
* **b**: 9
* **c**: 12
* **d**: 13
* **e**: 16

Steps to build the Huffman Tree:

1. Place each character and its frequency in a min-heap.
2. Extract the two nodes with the smallest frequencies (a: 5, b: 9) and create a new node with frequency 14 (5 + 9).
3. Repeat by combining nodes with the next smallest frequencies until only one node remains.

The tree might look something like this (in a simplified format):

scss

Copy code

(55)

/ \

(25) e:16

/ \

c:12 d:13

Each character now has a unique binary code based on its position in the Huffman Tree.

### 4. Why is Huffman coding considered lossless compression?

Huffman coding is considered **lossless compression** because it enables the exact reconstruction of the original data from the compressed data without any loss of information. This is possible because:

1. **Unique Prefix Property**: Each character is assigned a unique binary code, and no code is a prefix of another. This allows the encoded data to be unambiguously decoded.
2. **Deterministic Encoding and Decoding**: The process of encoding and decoding is deterministic. Given the same character frequencies, the Huffman Tree (and thus the encoding) is identical, allowing the original data to be precisely reconstructed.
3. **No Data Loss**: Unlike lossy compression (e.g., JPEG for images or MP3 for audio), where data is permanently discarded to reduce file size, Huffman coding retains all original information by only optimizing the binary representation.