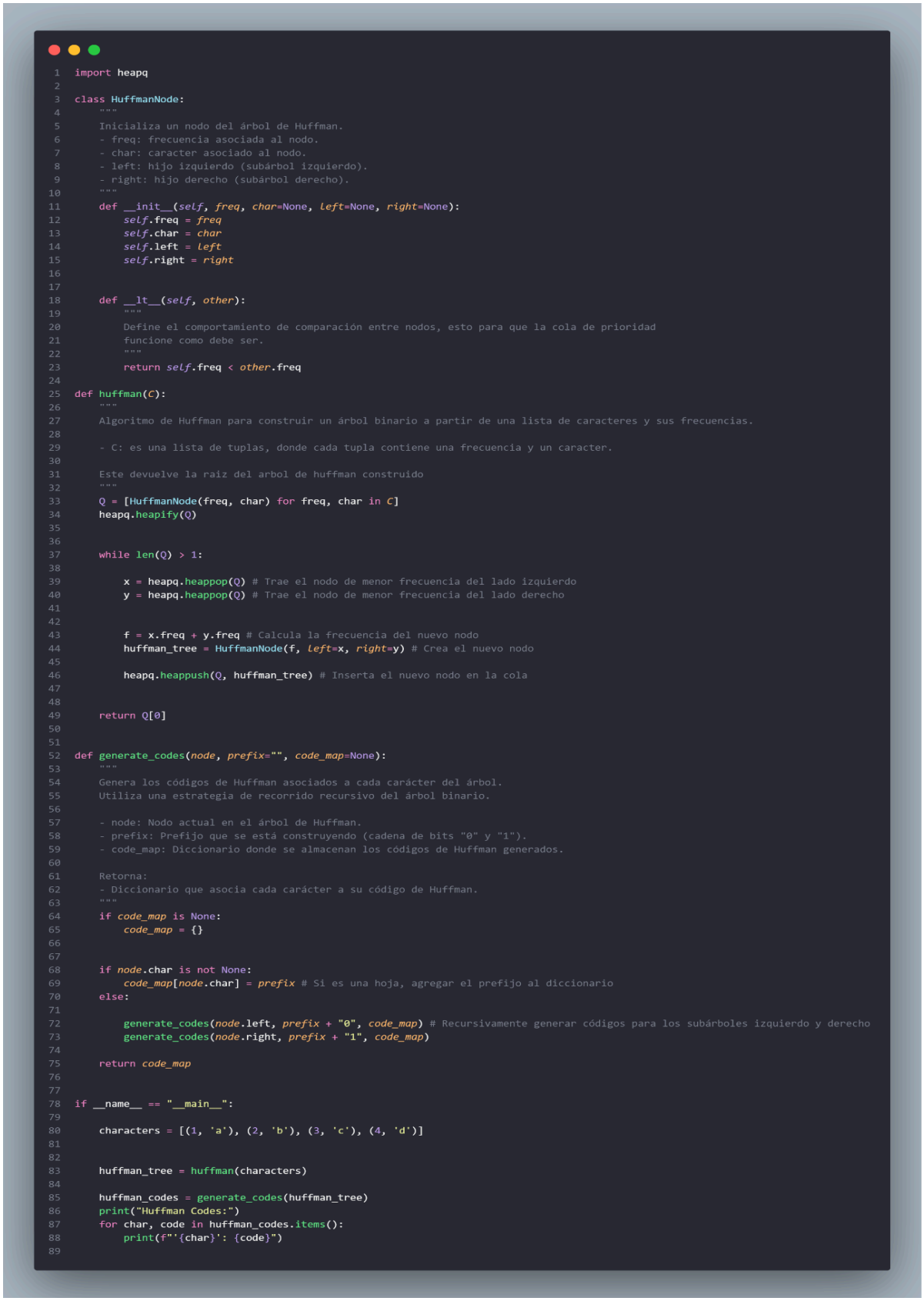
Nombre : Jesus David Gelves Cajiao - 30000098650

Huffman código Voraz



```python
import heapq

class HuffmanNode:
    """
    Inicializa un nodo del árbol de Huffman.
    - freq: frecuencia asociada al nodo.
    - char: caracter asociado al nodo.
    - left: hijo izquierdo (subárbol izquierdo).
    - right: hijo derecho (subárbol derecho).
    """
    def __init__(self, freq, char=None, left=None, right=None):
        self.freq = freq
        self.char = char
        self.left = left
        self.right = right


    def __lt__(self, other):
        """
        Define el comportamiento de comparación entre nodos, esto para que la cola de prioridad
        funcione como debe ser.
        """
        return self.freq < other.freq

def huffman(C):
    """
    Algoritmo de Huffman para construir un árbol binario a partir de una lista de caracteres y sus frecuencias.

    - C: es una lista de tuplas, donde cada tupla contiene una frecuencia y un caracter.

    Este devuelve la raiz del arbol de huffman construido
    """
    Q = [HuffmanNode(freq, char) for freq, char in C]
    heapq.heapify(Q)


    while len(Q) > 1:

        x = heapq.heappop(Q) # Trae el nodo de menor frecuencia del lado izquierdo
        y = heapq.heappop(Q) # Trae el nodo de menor frecuencia del lado derecho


        f = x.freq + y.freq # Calcula la frecuencia del nuevo nodo
        huffman_tree = HuffmanNode(f, left=x, right=y) # Crea el nuevo nodo

        heapq.heappush(Q, huffman_tree) # Inserta el nuevo nodo en la cola


    return Q[0]


def generate_codes(node, prefix="", code_map=None):
    """
    Genera los códigos de Huffman asociados a cada carácter del árbol.
    Utiliza una estrategia de recorrido recursivo del árbol binario.

    - node: Nodo actual en el árbol de Huffman.
    - prefix: Prefijo que se está construyendo (cadena de bits "0" y "1").
    - code_map: Diccionario donde se almacenan los códigos de Huffman generados.

    Retorna:
    - Diccionario que asocia cada carácter a su código de Huffman.
    """
    if code_map is None:
        code_map = {}


    if node.char is not None:
        code_map[node.char] = prefix # Si es una hoja, agregar el prefijo al diccionario
    else:

        generate_codes(node.left, prefix + "0", code_map) # Recursivamente generar códigos para los subárboles izquierdo y derecho
        generate_codes(node.right, prefix + "1", code_map)

    return code_map


if __name__ == "__main__":

    characters = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]


    huffman_tree = huffman(characters)

    huffman_codes = generate_codes(huffman_tree)
    print("Huffman Codes:")
    for char, code in huffman_codes.items():
        print(f"'{char}': {code}")

```

Análisis

Comparación con los otros enfoques:

Este se puede comparar con un algoritmo no voraz donde este es considerado un enfoque de fuerza bruta, en este caso se probaron todas las combinaciones para llegar a la solución más favorable o óptima. En este enfoque el costo sería de $O(n^2)$ y sería un enfoque que no es viable por tanto consumo de recursos.

Siguiendo con divide y vencerás, el algoritmo de huffman no se adapta a la estrategia mas que todo porque las combinaciones de nodos dependen mucho unas de otras y no es posible hacer una división en subproblemas.

Y por último programación dinámica, esta tampoco podría ser de ayuda para el algoritmo ya que no se necesita de cálculos que se almacenen y reutilizarlos.

Complejidad:

El primer paso en el algoritmo de Huffman es crear un monto o heap usando la librería que tiene python con los nodos iniciales. Este paso tiene una complejidad de $O(n)$, donde n es el número de caracteres, debido a heapify, que organiza los nodos de manera más rápida.

Una vez que el monto está listo, el algoritmo de Huffman comienza a combinar nodos. Cada combinación trae los dos nodos de menor frecuencia y la creación de un nuevo nodo con la suma de sus frecuencias. Esta operación se realiza $n - 1$ veces, y cada extracción y reinserción en el monto tiene un costo de $O(log\ n)$, lo que resulta en una complejidad total de $O(n\ log\ n)$ para esta fase.

Ya al finalizar y cuando ya esté construido el árbol de Huffman, se genera el código de cada carácter.

En conjunto, la complejidad total del algoritmo de Huffman es $O(n\ log\ n)$.