UNIVERSIDAD DE
SAN BUENAVENTURA
COLOMBIA

Universidad de San Buenaventura

Facultad de Ingeniería (Cali)

Ingeniería de Sistemas

Algoritmo voraz – Código de Huffman

Presenta:

Marlon Daniel Mora Restrepo

mdmorar@correo.usbcali.edu.co

Código: 30000087544

Asignatura**:** Análisis de algoritmos

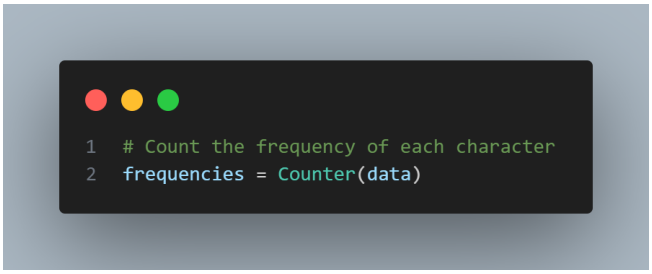
Docente**:** Carlos Mario Paredes

17 de noviembre de 2024

Análisis del algoritmo

Cálculo de frecuencias

En el código implementado, lo primero que se hace es contar las frecuencias de cada carácter de la siguiente forma:



```
1  # Count the frequency of each character
2  frequencies = Counter(data)
```

Esta clase recorre el texto completamente de carácter en carácter, por lo que su complejidad es $O(n)$, donde n es la longitud del texto.

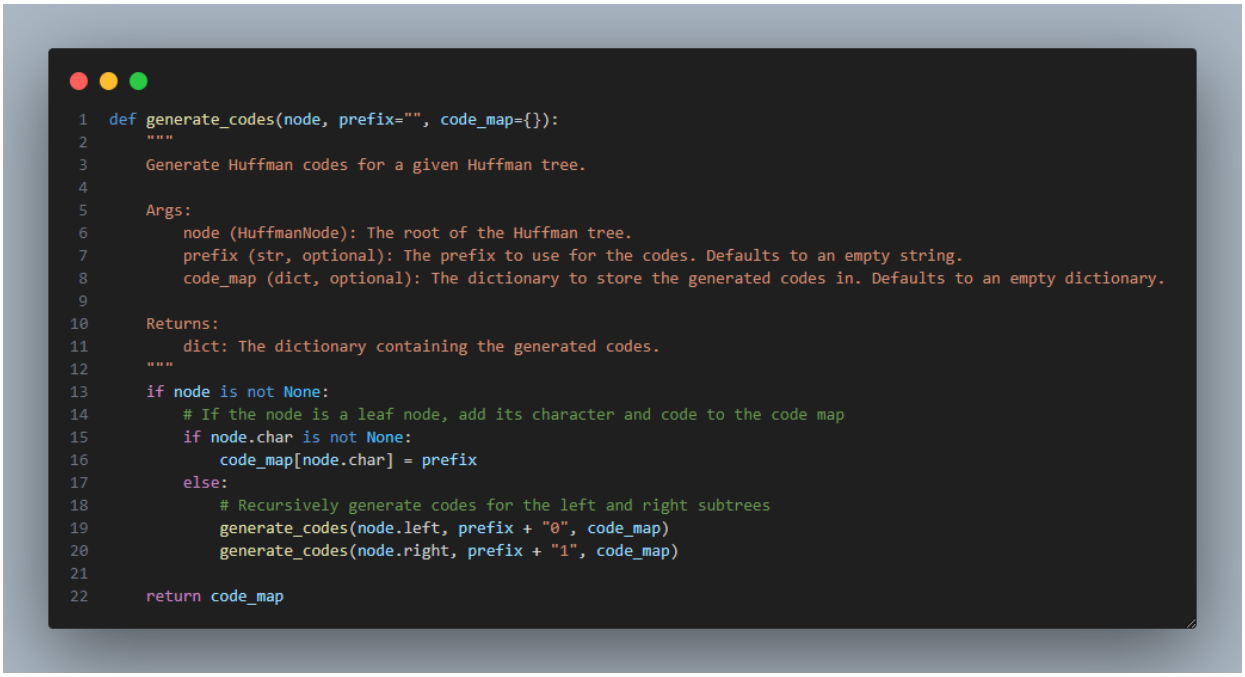
Construcción del árbol de Huffman

La construcción del árbol utiliza una cola de prioridad (heap), donde se realizan inserciones y extracciones basadas en las frecuencias. La inicialización del Heap (líneas 12 y 13) tiene una complejidad lineal $O(k)$, donde k es el número de caracteres únicos. En la combinación de los nodos, se extraen y se insertan dos nodos en cada operación, esto toma $O(\log k)$, por lo que la complejidad total de esta función de la función es $O(k \log k)$.

```
1  def build_huffman_tree(frequencies):
2      """
3      Build a Huffman tree from a given set of frequencies.
4
5      Args:
6          frequencies (dict): A dictionary mapping characters to their frequencies.
7
8      Returns:
9          HuffmanNode: The root of the Huffman tree.
10     """
11     # Create a priority queue to store the nodes
12     heap = [HuffmanNode(char, freq) for char, freq in frequencies.items()]
13     heapq.heapify(heap)
14
15     # Iterate until there is only one node left
16     while len(heap) > 1:
17         # Get the two nodes with the lowest frequencies
18         left = heapq.heappop(heap)
19         right = heapq.heappop(heap)
20
21         # Create a new node with the sum of the frequencies
22         merged = HuffmanNode(None, left.freq + right.freq)
23
24         # Assign the characters to the new node
25         merged.left = left
26         merged.right = right
27
28         # Add the new node to the priority queue
29         heapq.heappush(heap, merged)
30
31     return heap[0]  # Return the root of the Huffman tree
```

Generación de códigos

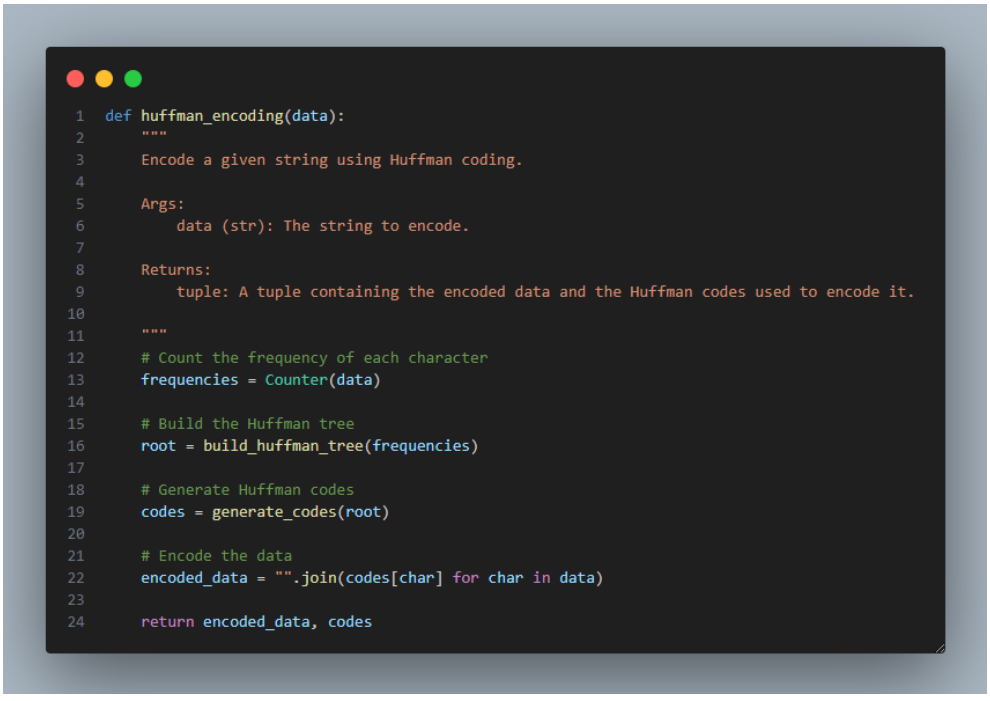
Aquí se recorre el árbol recursivamente para asignar los códigos. El árbol tiene k nodos y cada nodo se visita una vez, por lo que su complejidad es de $O(k)$.



```python
def generate_codes(node, prefix="", code_map={}):
    """
    Generate Huffman codes for a given Huffman tree.

    Args:
        node (HuffmanNode): The root of the Huffman tree.
        prefix (str, optional): The prefix to use for the codes. Defaults to an empty string.
        code_map (dict, optional): The dictionary to store the generated codes in. Defaults to an empty dictionary.

    Returns:
        dict: The dictionary containing the generated codes.
    """
    if node is not None:
        # If the node is a leaf node, add its character and code to the code map
        if node.char is not None:
            code_map[node.char] = prefix
        else:
            # Recursively generate codes for the left and right subtrees
            generate_codes(node.left, prefix + "0", code_map)
            generate_codes(node.right, prefix + "1", code_map)

    return code_map
```

Codificación del texto

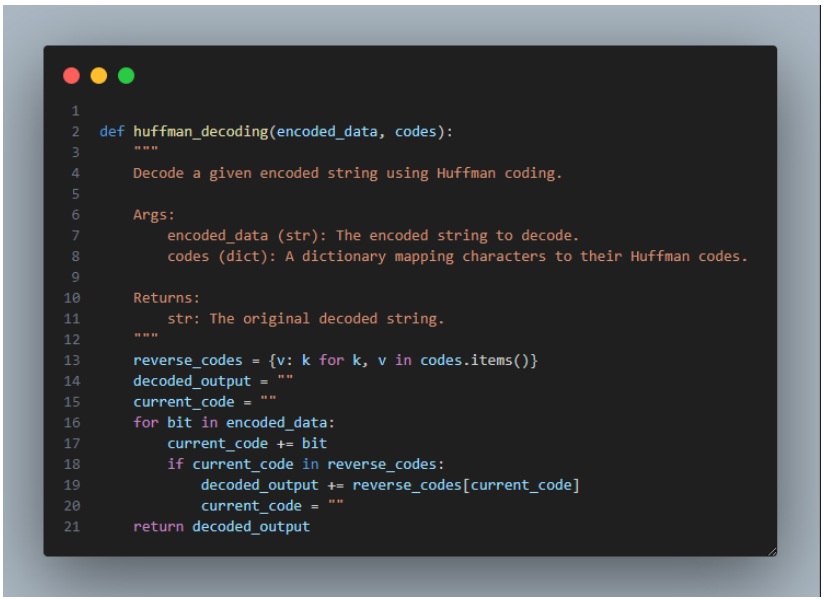
En esta parte ser recorre el texto y se reemplaza cada carácter por su código de Huffman, por lo que su complejidad es de $O(n)$, ya que se recorre el texto original.



```python
def huffman_encoding(data):
    """
    Encode a given string using Huffman coding.

    Args:
        data (str): The string to encode.

    Returns:
        tuple: A tuple containing the encoded data and the Huffman codes used to encode it.

    """
    # Count the frequency of each character
    frequencies = Counter(data)

    # Build the Huffman tree
    root = build_huffman_tree(frequencies)

    # Generate Huffman codes
    codes = generate_codes(root)

    # Encode the data
    encoded_data = "".join(codes[char] for char in data)

    return encoded_data, codes
```

Decodificación de texto

Para la reconstrucción del texto original, se recorre el texto codificado y ser usa un mapa invertido para construir los caracteres. Sea L la longitud del texto codificado, su complejidad será de $O(L)$.



```python

def huffman_decoding(encoded_data, codes):
    """
    Decode a given encoded string using Huffman coding.

    Args:
        encoded_data (str): The encoded string to decode.
        codes (dict): A dictionary mapping characters to their Huffman codes.

    Returns:
        str: The original decoded string.
    """
    reverse_codes = {v: k for k, v in codes.items()}
    decoded_output = ""
    current_code = ""
    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_codes:
            decoded_output += reverse_codes[current_code]
            current_code = ""
    return decoded_output
```

Complejidad total

La complejidad de todo el algoritmo dependerá del largo de la cadena de entrada *(n)* y el número de caracteres únicos *(k)*, por lo que siendo la construcción del árbol $O(k \log k)$, mientras que la codificación del texto es $O(n)$, La complejidad general es de $O(n + k \log k)$.