A logo of a university

Description automatically generated

International University of Information Technology

Department of Computer Engineering

**Laborotoy Work №5**

Complete by a student of the group: Ospan Ramazan it2-2310

Checked by senior lector of the Department of Computer Engineering:

Daurenbaeva Nurkamilya

Almaty 2024

**Huffman Algorithm**

The **Huffman Algorithm** is one of the most well-known methods of lossless data compression. It is used for efficient encoding of characters based on their frequency in the text.

**1. Main Idea of the Algorithm**

The Huffman algorithm minimizes the average length of codes for characters in a text. Characters that appear more frequently receive **shorter codes**, while less frequent characters receive **longer codes**. This method is called **prefix-free coding** because no code is a prefix of another code.

**2. Steps of the Huffman Algorithm**

1. **Counting character frequencies**: In the first step, the algorithm counts how often each character appears in the text. This helps to determine which characters are more frequent and which are rarer.
2. **Building the Huffman Tree**:
   * Each character is represented by a node, and the frequency of the character becomes the weight of the node.
   * The two nodes with the lowest frequency are combined into a new node, whose weight is the sum of the two nodes.
   * This process continues until there is only one node left, which becomes the root of the Huffman tree.
3. **Assigning codes to characters**: The Huffman tree is used to assign codes to the characters. By moving from the root of the tree to the leaves (characters), a binary code is assigned: moving left is 0 and moving right is 1.
4. **Encoding the text**: Each character in the text is replaced by its unique code, forming the encoded text.
5. **Decoding the text**: The Huffman tree is used to restore the original text. Each sequence of bits corresponds to a specific character, allowing the original text to be reconstructed.

**3. Example**

Let’s consider the string "ABABACD". The character frequencies are:

* A — 3 times,
* B — 2 times,
* C — 1 time,
* D — 1 time.

The Huffman algorithm will create the following codes:

* A = 0,
* B = 10,
* C = 110,
* D = 111.

The encoded text: 0100100110111.

**4. Advantages of the Huffman Algorithm**

* **Compression efficiency**: The more frequently a character appears, the shorter its code, which reduces the overall size of the data.
* **Prefix-free codes**: The algorithm ensures that no code is a prefix of another, preventing any ambiguity during decoding.
* **Real-world applications**: Huffman encoding is used in file formats such as ZIP, JPEG, and in data transmission systems.

**5. Applications**

The Huffman algorithm is widely used in data compression (e.g., ZIP archives, JPEG images) and is effective for texts with varying character frequencies.

In conclusion, the Huffman algorithm is an optimal way of encoding characters based on their frequency, significantly reducing the size of data without any loss of information.

Algorithm

# функция для вычисления частоты символов

def calculateThisFrequencies(thisText):

    thisFrequencies = {}

    for thisChar in thisText:

        if thisChar in thisFrequencies:

            thisFrequencies[thisChar] += 1  # увеличиваем частоту есл символ уже есть

        else:

            thisFrequencies[thisChar] = 1  # если нет, ставим 1 как частоту

    return thisFrequencies

# функция для постоения дерева хаффмана

def buildThisHuffmanTree(thisFrequencies):

    # создаем список из частот и символов

    thisTree = [[thisWeight, [thisChar, ""]] for thisChar, thisWeight in thisFrequencies.items()]

    # пока в дереве больше чем 1 элемент

    while len(thisTree) > 1:

        # сортируем по частоте

        thisTree = sorted(thisTree, key=lambda x: x[0])

        # извлекаем два наименьших элемента

        thisLeft = thisTree.pop(0)  # левая часть, самый маленький элемент

        thisRight = thisTree.pop(0)  # правая часть, второй самый маленький

        # присваеваем '0' для левого и '1' для правого

        for pair in thisLeft[1:]:

            pair[1] = '0' + pair[1]

        for pair in thisRight[1:]:

            pair[1] = '1' + pair[1]

        # добавляем новый узел в дерево, объединив два самых маленьких

        thisTree.append([thisLeft[0] + thisRight[0]] + thisLeft[1:] + thisRight[1:])

    # возвращаем отсортированные коды символов

    return sorted(thisTree[0][1:], key=lambda p: (len(p[-1]), p))

# функция для кодирования текста с помощью дерева хаффмана

def thisHuffmanEncode(thisText, thisHuffmanTree):

    thisHuffmanCode = {thisChar: thisCode for thisChar, thisCode in thisHuffmanTree}

    # заменяем символы на коды

    thisEncodedText = ''.join(thisHuffmanCode[thisChar] for thisChar in thisText)

    return thisEncodedText

# функция для декодирования закодированного текста

def thisHuffmanDecode(thisEncodedText, thisHuffmanTree):

    thisReverseCode = {thisCode: thisChar for thisChar, thisCode in thisHuffmanTree}

    thisDecodedText = ""

    thisTempCode = ""

    # перебераем каждый бит закодированного текста

    for thisBit in thisEncodedText:

        thisTempCode += thisBit

        if thisTempCode in thisReverseCode:

            thisDecodedText += thisReverseCode[thisTempCode]

            thisTempCode = ""

    return thisDecodedText

# функция для расчета коэффициента сжатия

def calculateThisCompressionRatio(thisOriginalText, thisEncodedText):

    thisOriginalSize = len(thisOriginalText) \* 8  # исходный размер (1 символ = 8 бит)

    thisCompressedSize = len(thisEncodedText)  # размер сжатого текста в битах

    return thisOriginalSize / thisCompressedSize

# основная функция для работы с вводом и выводом

def main():

    thisText = input("введите текст для сжатия: ")

    # вычисляем частоты символов и строим дерево хаффмана

    thisFrequencies = calculateThisFrequencies(thisText)

    thisHuffmanTree = buildThisHuffmanTree(thisFrequencies)

    # кодируем текст

    thisEncodedText = thisHuffmanEncode(thisText, thisHuffmanTree)

    print(f"сжатый текст: {thisEncodedText}")

    print(f"коды символов: {thisHuffmanTree}")

    # декодируем текст

    thisDecodedText = thisHuffmanDecode(thisEncodedText, thisHuffmanTree)

    print(f"распакованный текст: {thisDecodedText}")

    # проверка, совпадают ли оригинальный и распакованный тексты

    if thisText == thisDecodedText:

        print("распаковка удалась, текст совпадает.")

    else:

        print("ошибка: распакованный текст не совпадает с исходным.")

    # расчет и вывод коэффициента сжатия

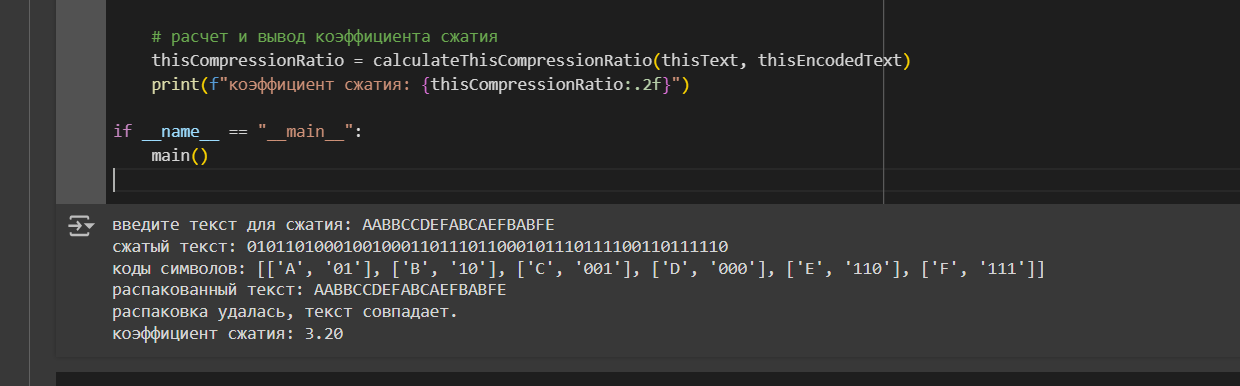
    thisCompressionRatio = calculateThisCompressionRatio(thisText, thisEncodedText)

    print(f"коэффициент сжатия: {thisCompressionRatio:.2f}")

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

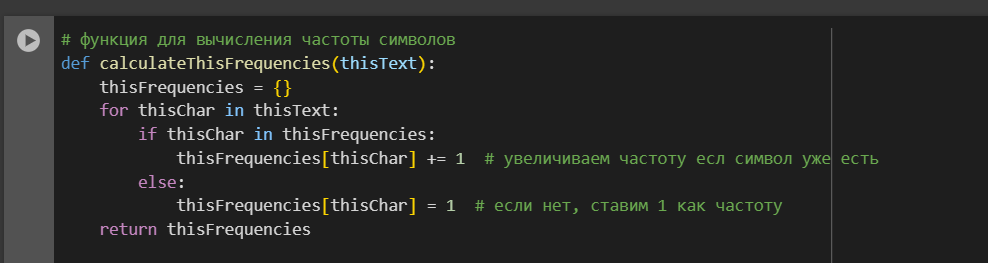
    main()

**Test:**



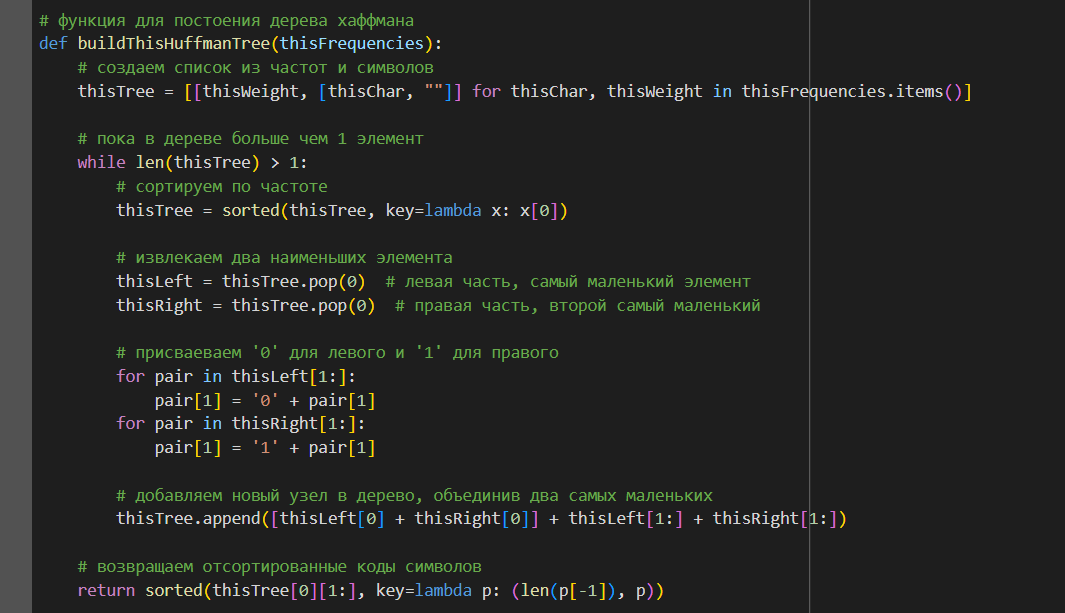
**Explanation:**

* 1. **Character Frequency Calculation**

****

* This function takes the input text (thisText) and calculates how many times each character appears in the text.
* It uses a dictionary thisFrequencies where the key is the character, and the value is its frequency.
* If the character is already in the dictionary, its frequency is increased by 1. Otherwise, the character is added to the dictionary with an initial count of 1.
* The function returns the thisFrequencies dictionary containing the frequency of each character.

2. **Building the Huffman Tree**



 This function builds the Huffman tree based on the character frequencies provided by the previous function.

 The thisTree variable is a list of lists, where each inner list represents a character and its frequency, along with its Huffman code (initialized as an empty string).

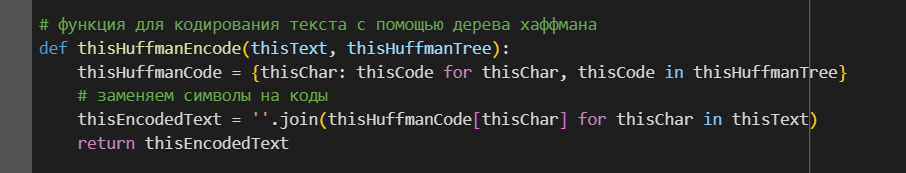
 The algorithm works by repeatedly:

1. Sorting the thisTree list by frequency.
2. Taking the two lowest frequency elements (thisLeft and thisRight).
3. Assigning a 0 to the left element and a 1 to the right element.
4. Combining their frequencies into a new node and adding it back into the tree.

 The process continues until only one node remains in the tree, representing the root of the Huffman tree.

 Finally, the function returns the Huffman codes for each character.

3. **Encoding the Text**



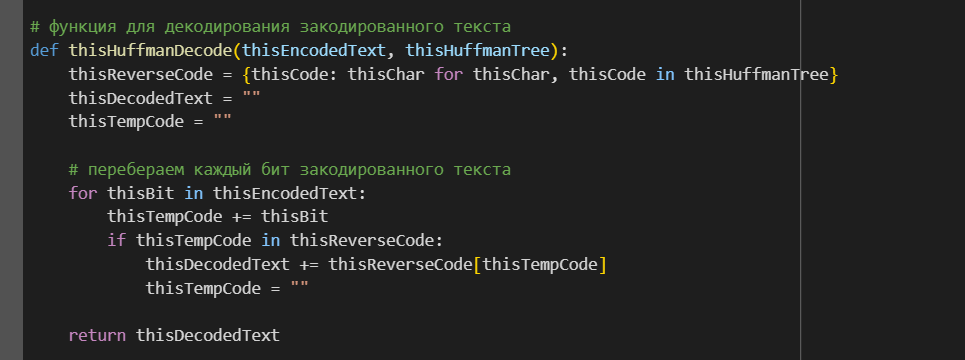
 This function encodes the input text (thisText) using the Huffman codes generated by the buildThisHuffmanTree function.

 It creates a dictionary thisHuffmanCode where the key is the character and the value is the corresponding Huffman code.

 The function then goes through each character in the input text and replaces it with its corresponding Huffman code, forming the encoded text (thisEncodedText).

 The encoded text is returned as a single string of bits.

4. **Decoding the Encoded Text**



 This function decodes the previously encoded text (thisEncodedText) using the Huffman tree.

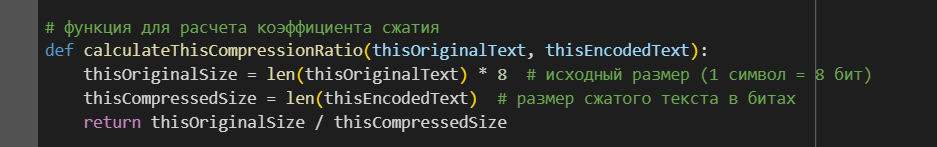
 A dictionary thisReverseCode is created, where the key is the Huffman code, and the value is the corresponding character (this is the reverse of the encoding process).

 The function reads each bit of the encoded text and builds up the Huffman code (thisTempCode) until it finds a complete code that matches a character in thisReverseCode.

 The decoded character is added to thisDecodedText, and the process continues until the entire text is decoded.

 The function returns the decoded text.

5. **Calculating the Compression Ratio**



 This function calculates the compression ratio, which is a measure of how effective the Huffman compression is.

 The original size of the text is calculated by multiplying the number of characters by 8 (since each character is 8 bits).

 The compressed size is simply the length of the encoded text (in bits).

 The compression ratio is calculated by dividing the original size by the compressed size. A higher ratio means better compression.

6. **Main Function**



 This is the main function that ties everything together.

 It takes user input (thisText), calculates the character frequencies, builds the Huffman tree, and then encodes the text.

 It displays the encoded text and the Huffman codes.

 The function also decodes the encoded text and checks if the decoded text matches the original input.

 Finally, it calculates the compression ratio and displays it.

Resume:

 **Character Frequency Calculation**: Counts how often each character appears in the input text.

 **Building the Huffman Tree**: Constructs a tree based on character frequencies, assigning binary codes to each character.

 **Encoding the Text**: Converts the input text into a compressed binary form using the Huffman codes.

 **Decoding the Text**: Converts the encoded binary text back into the original text using the Huffman tree.

 **Compression Ratio**: Measures the efficiency of the compression by comparing the size of the original text and the compressed text.

In conclusion:

The main steps involved in the Huffman algorithm include calculating character frequencies, building the Huffman tree, encoding the text, decoding the encoded text, and calculating the compression ratio to assess the efficiency of the compression. The algorithm ensures that no code is a prefix of another, which makes it easy to decode the compressed text without ambiguity. By comparing the original text size to the compressed text size, we can observe the significant reduction in data size, which proves the effectiveness of Huffman coding.