

## ，实验四 Huffman 编码的实现与压缩软件实验报告

学号： 21030031009 姓名： 惠欣宇 电话： 18149045867

本实验要求独立完成。实验报告正文要求采用小四号字，中文使用宋体，英文使用 Times New Roman。对于实验中出现的代码文字，可使用 Consolas 字体。实验报告要求提交 pdf 电子版。电子版文件命名格式要求为：学号-姓名.pdf。

### 1. 实验目的

- 1) 深入理解 Huffman 编码的设计理念，掌握 Huffman 编码、解码流程。
- 2) 使用 Python 语言编程实现 Huffman 编码对文件的压缩与解压缩。

### 2. 实验预备知识及实验要求

#### 2.1. 实验预备知识

- 1) Huffman 编码相关知识
- 2) Python 编程语言基础语法及常用数据结构。
- 3) 参考资料:  
HuffmanCoding.ppt

#### 2.2. 实验要求:

- 1) Huffman 编码及解码的算法思想（可用伪代码表述），须与上交的代码保持一致。

哈夫曼编码是一个通过哈夫曼树进行的一种编码，一般情况下，以字符：0 与 1 表示。编码的实现过程很简单，只要实现哈夫曼树，通过遍历哈夫曼树，这里我们从每一个叶子结点开始向上遍历，如果该结点为父节点的左孩子，则在字符串后面追加 0，如果为其右孩子，则在字符串后追加 1。结束条件为没有父节点。然后将字符串倒过来存入结点中。

哈夫曼编码使用变长编码表对源符号（如文件中的一个字母）进行编码，其中变长编码表是通过一种评估来源符号出现机率的方法得到的，出现机率高的字母使用较短的编码，反之出现机率低的则使用较长的编码，这便使编码之后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。

哈夫曼编码的具体步骤如下：

- ①将信源符号的概率按减小的顺序排队。
- ②把两个最小的概率相加，并继续这一步骤，始终将较高的概率分支放在右边，直到最后变成概率 1。
- ③画出由概率 1 处到每个信源符号的路径，顺序记下沿路径的 0 和 1，所得就是该符号的霍夫曼码字。
- ④将每对组合的左边一个指定为 0，右边一个指定为 1（或相反）。

2) 展示实验过程，可适当截图，并标注说明。

下面是本次实验使用到的模块：

`os` 模块提供的就是程序与操作系统进行交互的接口。通过使用 `os` 模块，一方面可以方便地与操作系统进行交互，另一方面还可以极大增强代码的可移植性，判断文件是否存在。

`pickle` 模块用来打包存储 `python` 字符类型和数据类型，该模块仅限在 `python` 中使用。把 `python` 对象直接保存到文件里，而不需要先把它们转化为字符串再保存，也不需要用底层的文件访问操作，直接把它们写入到一个二进制文件里。`pickle` 模块会创建一个 `python` 语言专用的二进制格式，不需要使用者考虑任何文件细节，它会帮你完成读写对象操作。

`struct` 模块作用是完成 `python` 数值和 C 语言结构体的 `python` 字符串形式间的转换。这可以用于处理存储在文件中或从网络连接中存储的二进制数据，以及其他数据源。

```
import os
import pickle # 可对Python的原生对象进行“打包存储”，此处用来存储全局的huffman_map
import struct # 用来存取二进制数据
```

下面是定义的结点类：

首先初始化左子树、右子树、字符与权值四个变量，权值便是每种字符在文章中出现的次数。然后再判断是否为叶子结点，若为叶子节点则得到一串编码。

```
class Node(object): # 定义结点类
    def __init__(self, symbol='', weight=0):
        self.left = None # 左子树
        self.right = None # 右子树
        self.symbol = symbol # 字符
        self.weight = weight # 权值

    1 usage
    def isLeaf(self): # 判断是否为叶子节点
        return not (self.left or self.right)
```

下面是一些全局变量（全局变量功能在注释中）：

```
huffman_map = {} # 存储字符与对应的编码
huffman_tree_root = Node() # Huffman树的根节点
letter_frequency = {} # 存储字符及其出现次数（权重）
```

下面是压缩文件的函数 `compress`、读取文件函数 `read_file`、得到 `huffman` 编码函数 `get_huffman_map`、编码函数 `encode` 与构建哈夫曼二叉树函数 `huffman_tree` 函数的代码思路：

然后验证构造的 `huffman` 二叉树是否正确，输出出现频率最高的三个字符及对应的编码。

```

def compress():
    """
    压缩文件
    """
    # 输入待编码文件地址
    file = input("Please input the file that you want to compress:")
    while not os.path.exists(file):
        print("The file that you input is not exists.")
        file = input("Please input again:")

    # 读取待编码文件，并统计letter_frequency。|
    ori_data = read_file(file)

    # 构建Huffman树
    huffman_tree()

    # 生成huffman_map，即每个字母与其Huffman编码的对应
    get_huffman_map(huffman_tree_root)

    # 输出出现频率最高的3个字符及其对应的Huffman编码，以验证编码的正确性。
    fre = sorted(letter_frequency.items(), key=lambda f: f[1], reverse=True)
    for i in range(0, 3):
        print(i, fre[i][0], fre[i][1], huffman_map.get(fre[i][0]))

    # 对文章进行编码，并写入新文件
    # 新文件命名方式为：源文件 A.txt -> 新文件 A_coded.txt
    new_file = file[: -4] + ".hfm"
    encode(ori_data, new_file)

    print("Finished compressing.")

```

首先输入一个文件名，使用 `os` 模块判断该文件是否存在，不存在的话重新输入文件名，存在则使用 `read_file` 函数使用 `utf-8` 的方式读取文件，并统计文件中每个字符出现的次数存入全局变量 `letter_frequency` 中，`read_file` 函数将文件中的内容存入 `text` 变量中，每次读取一行，读取结束后加入变量 `text` 中，一边读取一边统计每一个字符的出现次数，最后返回变量 `text`。

```

def read_file(path):
    """
    读取文件，并统计文件中每个字符出现的次数存入全局的Letter_frequency。
    :param path: 待读取文件
    :type path: str
    :return: 读取的文件内容
    :rtype: str
    """
    with open(path, 'r', encoding='utf-8') as f:
        text = ""
        for line in f.readlines():
            text += line
            for letter in line:
                if letter_frequency.get(letter):
                    letter_frequency[letter] += 1
                else:
                    letter_frequency[letter] = 1

    return text

```

紧接着调用 `huffman_tree` 函数构造 Huffman 二叉树，以嵌套的方式实现，并存入全局变量 `huffman_tree_root` 中。

`huffman_tree` 函数中先将所有节点存入临时列表 `nodes` 中，再把 `letter_frequency` 中的字符构造成结点，再将其加入列表中。

接下来构造二叉树，循环构造直到只剩下根结点，每一次构造会少一个结点。

循环中对结点以权值进行排序，排序完成后将权值最小的两个结点从列表中弹出，分别赋值给 `left` 与 `right`，构造出一个合并后的新结点，新结点的权值就两个结点的权值之和。

然后将原本的两个结点嵌套入新结点的左右子树中，最后将新结点压入列表中，继续进行上述循环。

在循环结束后，仅剩下一个根结点，如果根结点不为空，就将根结点深拷贝给全局变量 `huffman_tree_root`。

```
def huffman_tree(): # 构建哈夫曼树，
    """
    构建Huffman树，以嵌套的方式实现，并存入全局的huffman_tree_root中。
    :return: None
    """
    SIZE = len(letter_frequency)

    # 先将所有节点存入临时列表
    nodes = []
    for (letter, fre) in letter_frequency.items():
        nodes.append(Node(letter, fre))

    for _ in range(SIZE - 1):
        nodes.sort(key=(lambda n: n.weight)) # 以权值对节点进行排序
        left = nodes.pop(0) # 提取权值最小的两个节点
        right = nodes.pop(0)
        parent = Node('', left.weight + right.weight) # 创建一个新的节点
        parent.left = left # 将之前提取的两个权值最小的节点嵌套至新结点中
        parent.right = right
        nodes.append(parent) # 将新结点加入nodes

    if nodes:
        huffman_tree_root.left = nodes[0].left
        huffman_tree_root.right = nodes[0].right
        huffman_tree_root.symbol = nodes[0].symbol
        huffman_tree_root.weight = nodes[0].weight
```

构建好 huffman 二叉树之后，调用 `get_huffman_map` 函数得到每个字符对应的 huffman 编码并存入全局变量 `huffman_map` 中（编码原则为左 0 右 1）。`get_huffman_map` 函数采用递归的方式，每次传入一个子树，判断当前子树是否为叶子结点，说明已经找到一棵子树以及对应的编码，将其存入 `huffman_map` 变量中并且弹出该次递归，如果传入的子树不是叶子结点，则向左子树行进，再次递归调用，传入左子树的左子树，编码原则不变，左结点为空就遍历右结点，此时传入右子树，编码变为 1。

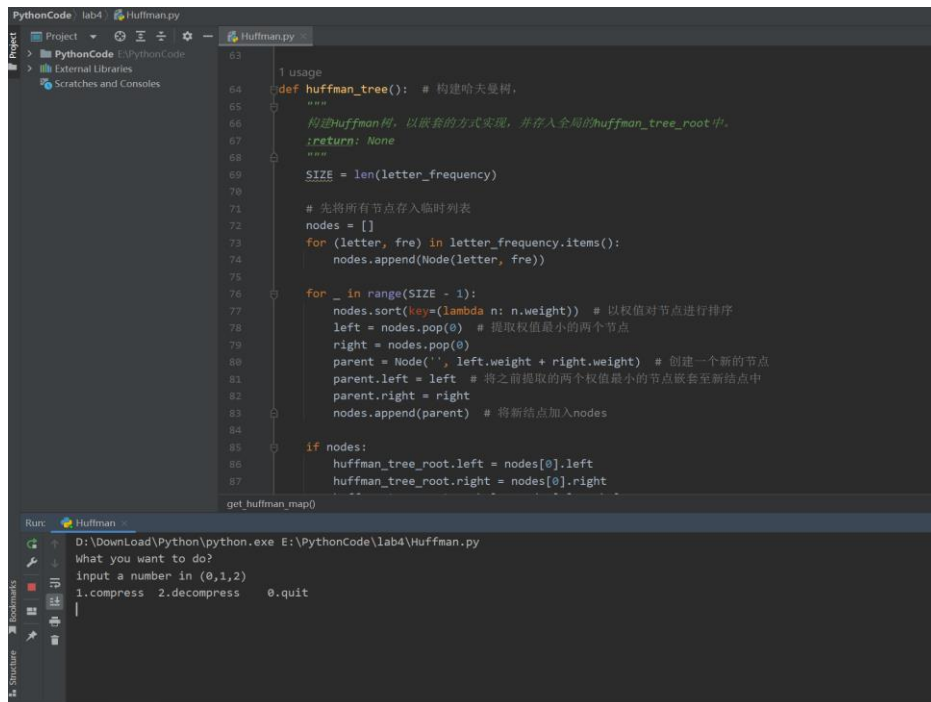
```
def get_huffman_map(tree, code=''):
    """
    递归调用自身，获取每个字母的对应的Huffman编码，存入全局的huffman_map。
    :param tree: Huffman树
    :type tree: Node
    :param code: Huffman编码，一串'0' '1'序列
    :type code: str
    :return: None
    """
    if tree.isLeaf():
        huffman_map[tree.symbol] = code
        return
    if tree.left:
        get_huffman_map(tree.left, code + '0')
    if tree.right:
        get_huffman_map(tree.right, code + '1')
```

最后将 `encode` 后的内容写入新的文件中，文件后缀名为 `.hfm`。对 `ori_data` 内容按照全局的 `huffman_map` 信息进行编码，并写入新的文件中。

```
def encode(ori_data, new_file):
    """
    对ori_data内容按照全局的huffman_map信息进行编码，并写入new_file中
    :param ori_data: 读取到的源文件的内容
    :type ori_data: str :param new_file: 新文件路径名称
    :type new_file: str :return: None
    """
    # 编码，将形成的Huffman编码先拼成一个字符串
    coded_data = ""
    for letter in ori_data:
        coded_data += huffman_map.get(letter)
    # 将全局的letter_frequency和编码后的coded_data转为二进制，并写入new_file。
    huffman_map_bytes = pickle.dumps(huffman_map)
    f = open(new_file, 'wb')
    # 存储全局的huffman_map
    f.write(struct.pack('!%ds' % (len(huffman_map_bytes)),
                       len(huffman_map_bytes), huffman_map_bytes))
    # 存储编码后数据coded_data
    # 8位（1字节）一组进行存储，先看看是不是长度是不是8的整数倍
    f.write(struct.pack('!B', len(coded_data) % 8)) # 存储最后不足一字节位数的二进制位数，让解码器知道有没有补0
    for index in range(0, len(coded_data), 8):
        if index + 8 < len(coded_data):
            # 将8位二进制字符串（占8字节）转换为1位十进制（占1字节）整形
            f.write(struct.pack('!B', int(coded_data[index:index+8], 2)))
        else:
            # 最后若干位（<=8）转换成一个十进制整形
            f.write(struct.pack('!B', int(coded_data[index:], 2)))
    f.close()
```

### 3) 实验结果

这里压缩与解压一份名为“車内で化粧をする女性（在车内化妆的女性）”的日语文章。



```
PythonCode lab4 Huffman.py
Project
  PythonCode E:\PythonCode
  External Libraries
  Scratches and Consoles
Huffman.py
63 1 usage
64 def huffman_tree(): # 构建哈夫曼树,
65     """
66     构建huffman树, 以嵌套的方式实现, 并存入全局的huffman_tree_root中.
67     :return: None
68     """
69     SIZE = len(letter_frequency)
70
71     # 先将所有节点存入临时列表
72     nodes = []
73     for (letter, fre) in letter_frequency.items():
74         nodes.append(Node(letter, fre))
75
76     for _ in range(SIZE - 1):
77         nodes.sort(key=(lambda n: n.weight)) # 以权值对节点进行排序
78         left = nodes.pop(0) # 提取权值最小的两个节点
79         right = nodes.pop(0)
80         parent = Node('', left.weight + right.weight) # 创建一个新的节点
81         parent.left = left # 将之前提取的两个权值最小的节点嵌套至新节点中
82         parent.right = right
83         nodes.append(parent) # 将新节点加入nodes
84
85     if nodes:
86         huffman_tree_root.left = nodes[0].left
87         huffman_tree_root.right = nodes[0].right
88
89     get_huffman_map()
```

Run Huffman

D:\Download\Python\python.exe E:\PythonCode\lab4\Huffman.py

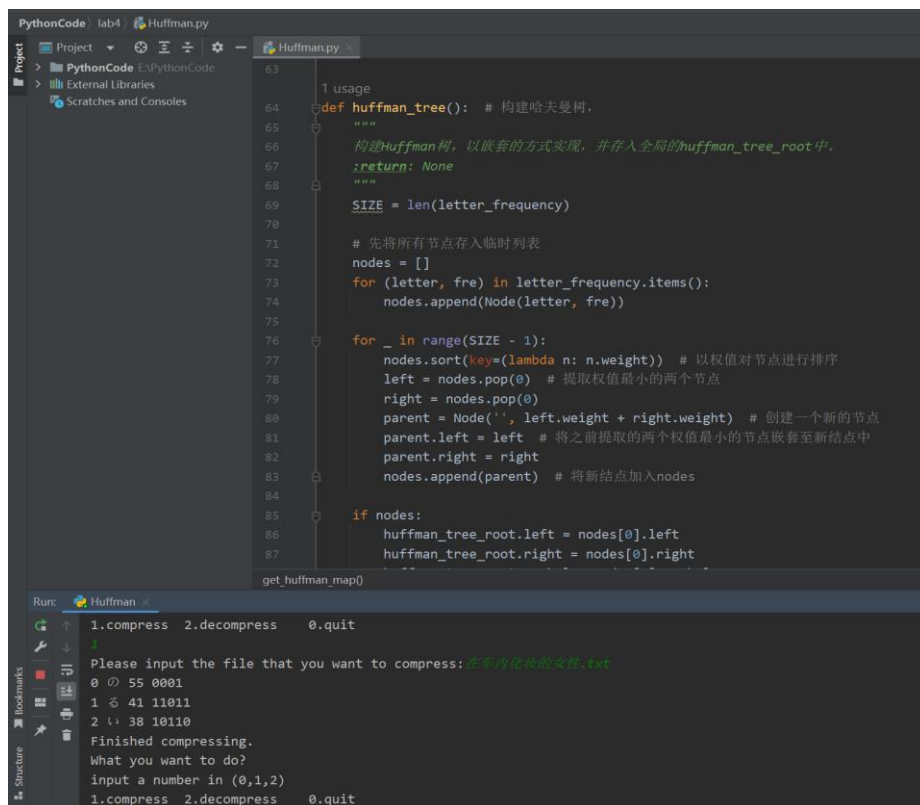
What you want to do?

input a number in (0,1,2)

1.compress 2.decompress 0.quit

|

调用 `compress` 函数对文件压缩完成，并输出出现频率最高的三个字符及其对应的编码：



```
PythonCode lab4 Huffman.py
Project
  PythonCode E:\PythonCode
  External Libraries
  Scratches and Consoles
Huffman.py
63 1 usage
64 def huffman_tree(): # 构建哈夫曼树,
65     """
66     构建huffman树, 以嵌套的方式实现, 并存入全局的huffman_tree_root中.
67     :return: None
68     """
69     SIZE = len(letter_frequency)
70
71     # 先将所有节点存入临时列表
72     nodes = []
73     for (letter, fre) in letter_frequency.items():
74         nodes.append(Node(letter, fre))
75
76     for _ in range(SIZE - 1):
77         nodes.sort(key=(lambda n: n.weight)) # 以权值对节点进行排序
78         left = nodes.pop(0) # 提取权值最小的两个节点
79         right = nodes.pop(0)
80         parent = Node('', left.weight + right.weight) # 创建一个新的节点
81         parent.left = left # 将之前提取的两个权值最小的节点嵌套至新节点中
82         parent.right = right
83         nodes.append(parent) # 将新节点加入nodes
84
85     if nodes:
86         huffman_tree_root.left = nodes[0].left
87         huffman_tree_root.right = nodes[0].right
88
89     get_huffman_map()
```

Run Huffman

1.compress 2.decompress 0.quit

1

Please input the file that you want to compress: 在车内化妆的女性.txt

0 0 55 0001

1 0 41 11011

2 0 38 10110

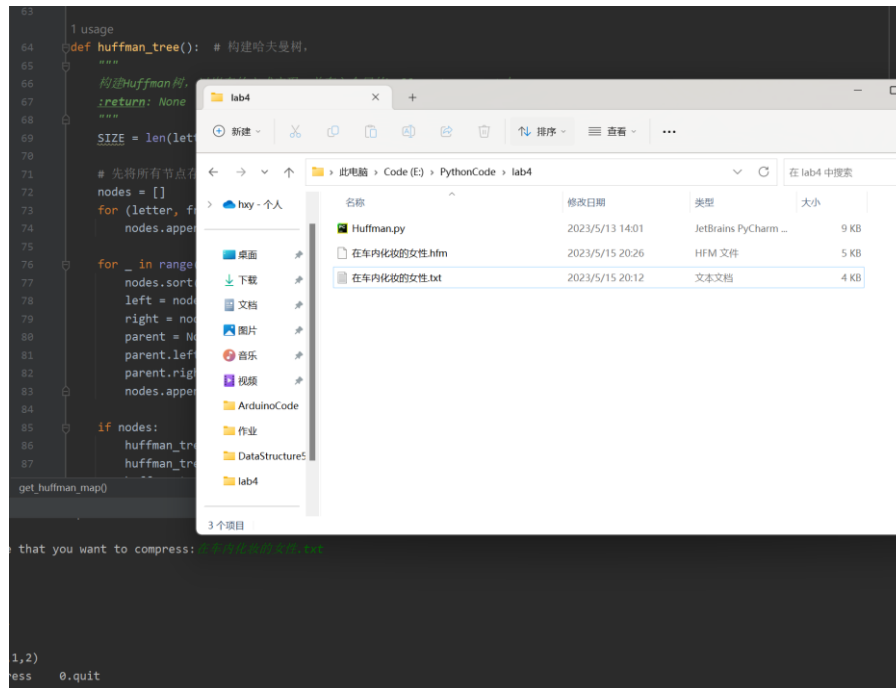
Finished compressing.

What you want to do?

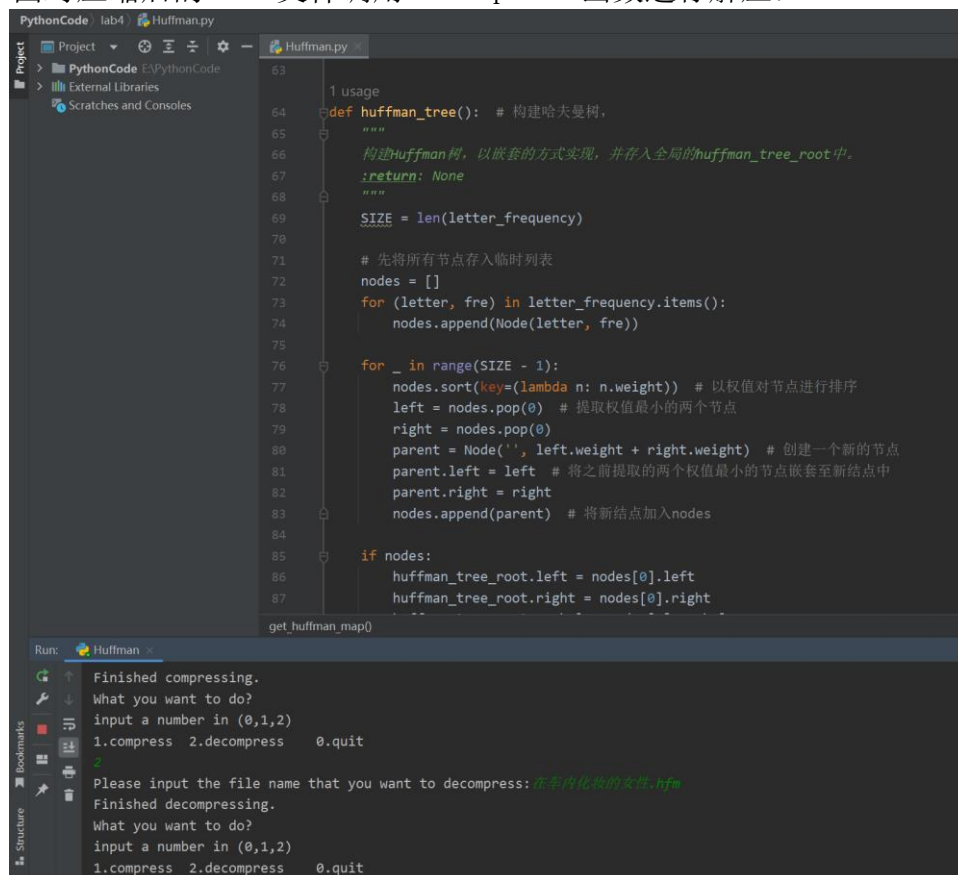
input a number in (0,1,2)

1.compress 2.decompress 0.quit

我们在项目文件夹中发现已经压缩后的.hfm文件，说明已经压缩完成：

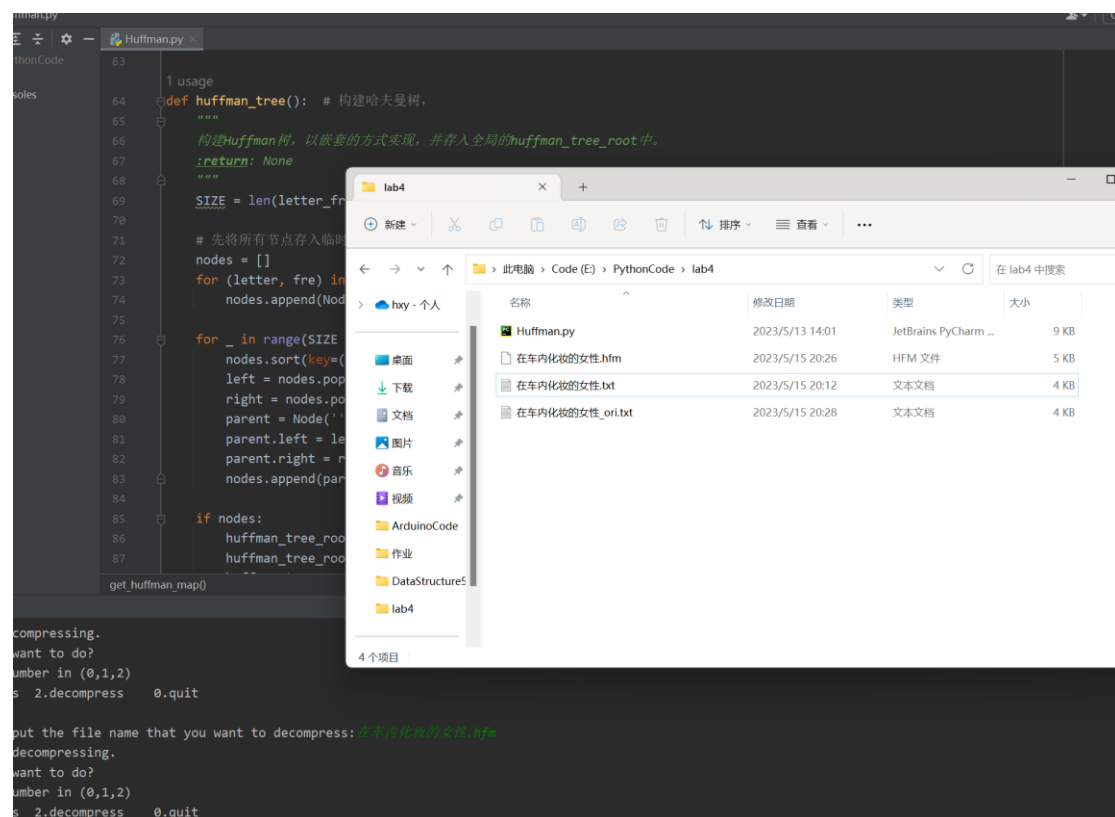


下面对压缩后的.hfm文件调用decompress函数进行解压：





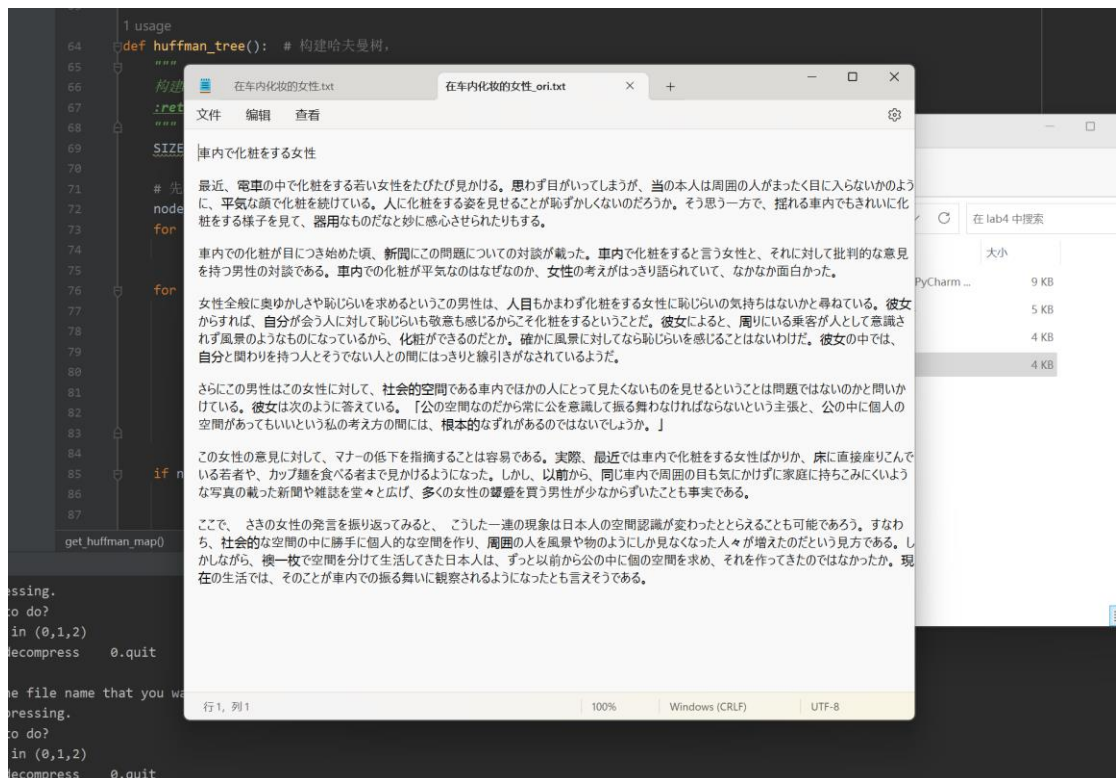
解压完成后我们在项目文件夹中发现\_ori.txt 文件，说明解压完成：



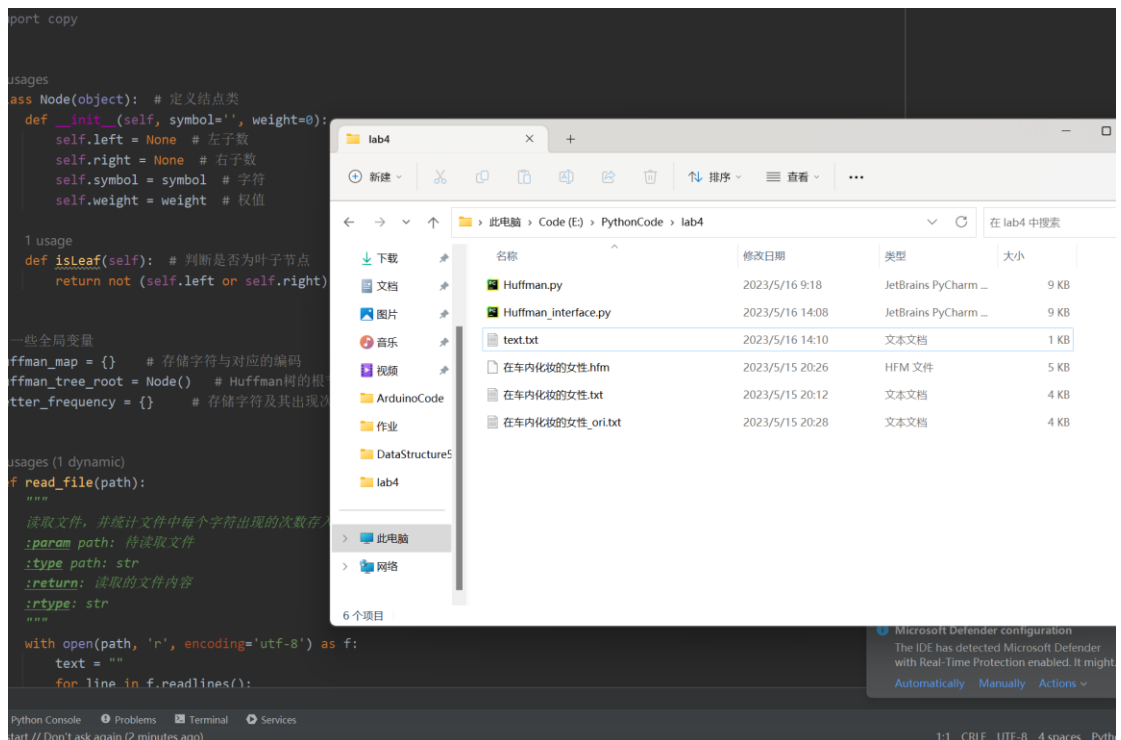
最后对比源文件与经过压缩形成.hfm 文件并解压回\_ori.txt 文件的内容，发现二者内容一致，本次实验成功：



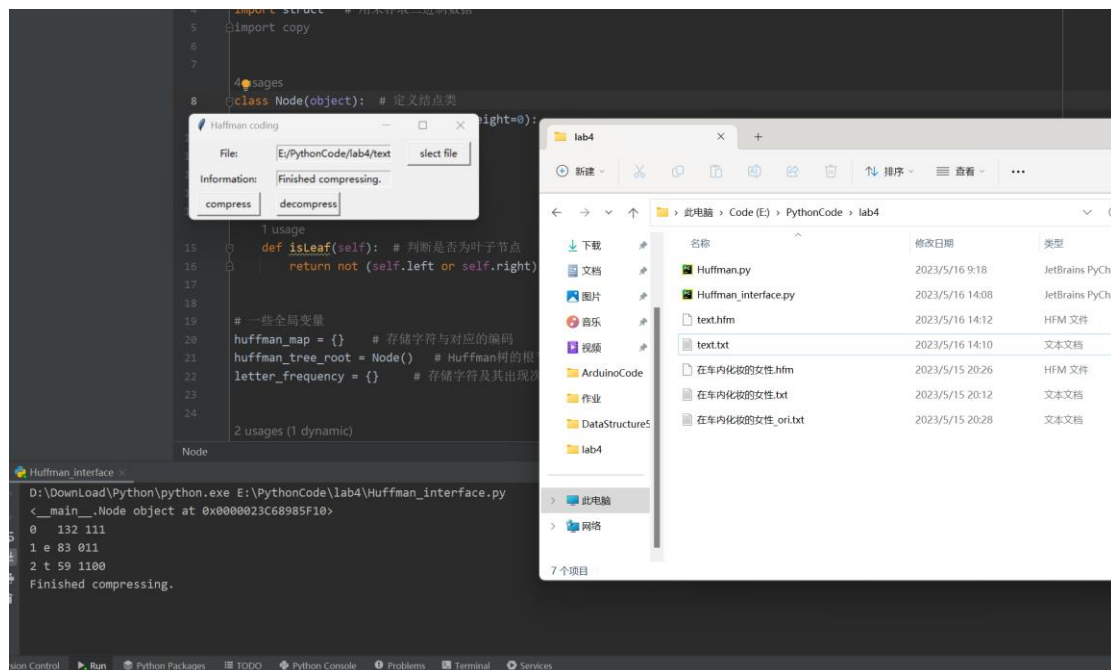




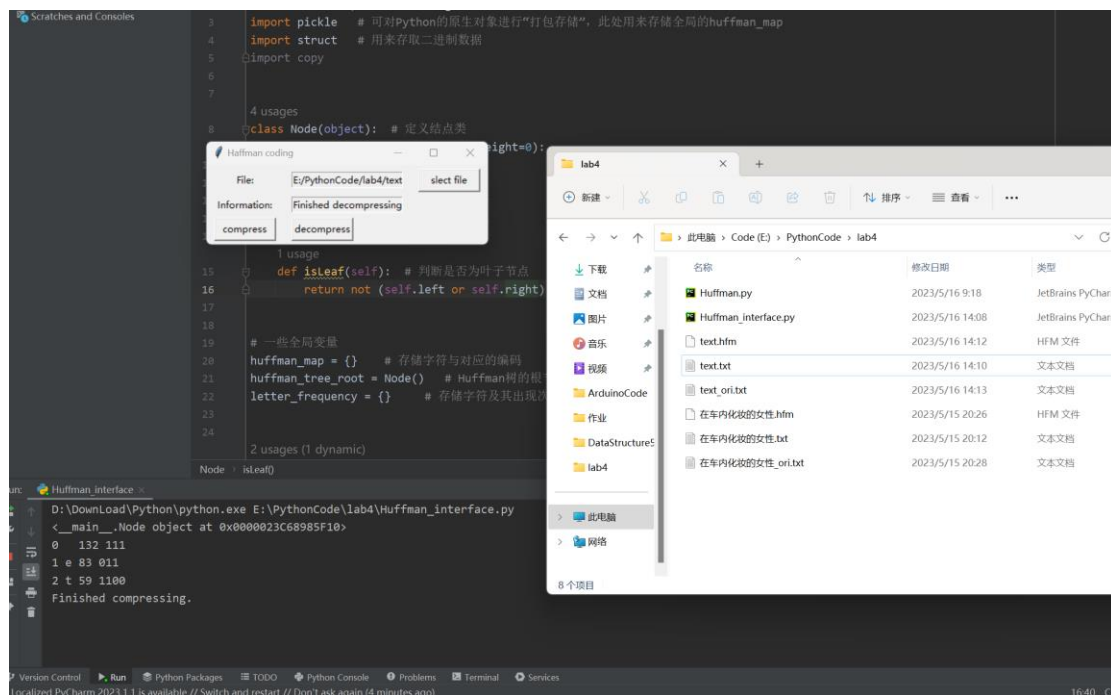
下面是实现界面交互的 Huffman 编解码程序的运行结果：  
编解码文件为 text.txt:



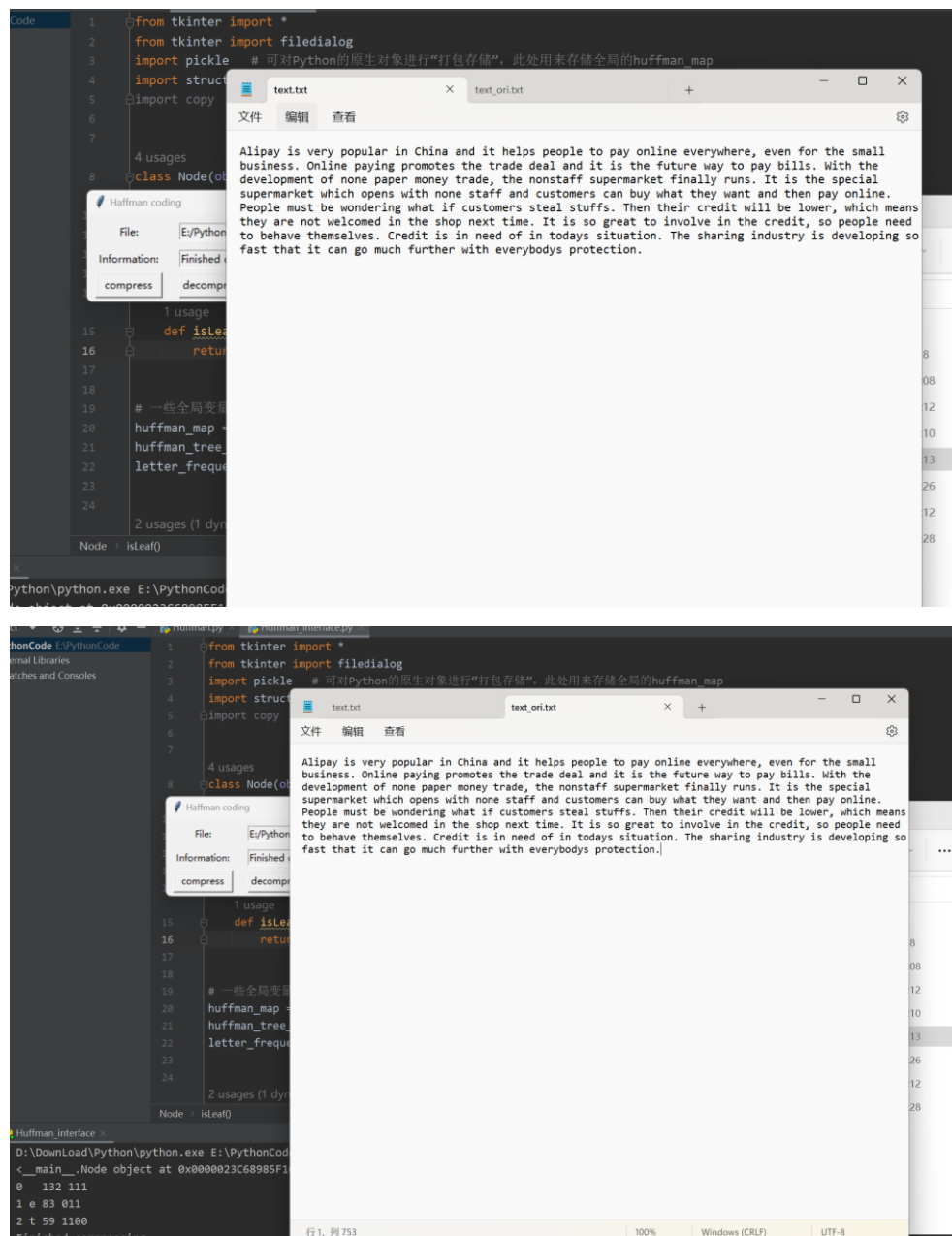
压缩 text.txt 得到 text.hfm 文件:



对 text.hfm 文件进行解码得到 text\_ori.txt:



源文件与解压文件进行对比：



#### 4) 总结分析。

加深了对哈夫曼树的理解，学习了利用贪心算法的思想创建哈夫曼树的方式。学习了哈夫曼树编码解码的流程，通过比较权值逐步构建一颗 Huffman 树，再由 Huffman 树进行编码、解码。学习到了哈夫曼树是一个最优编码问题，通过贪心选择保留了最优解。巩固了编码解码的知识。遇到过编码中文文件后编码太长的原因，因为中文字符的种类过多，而英文也就 26 个字母加一些其他的字符，因此使用哈夫曼树编码中文文件可能会得不偿失，导致文件体积进一步增大