**DAA Lab 5**

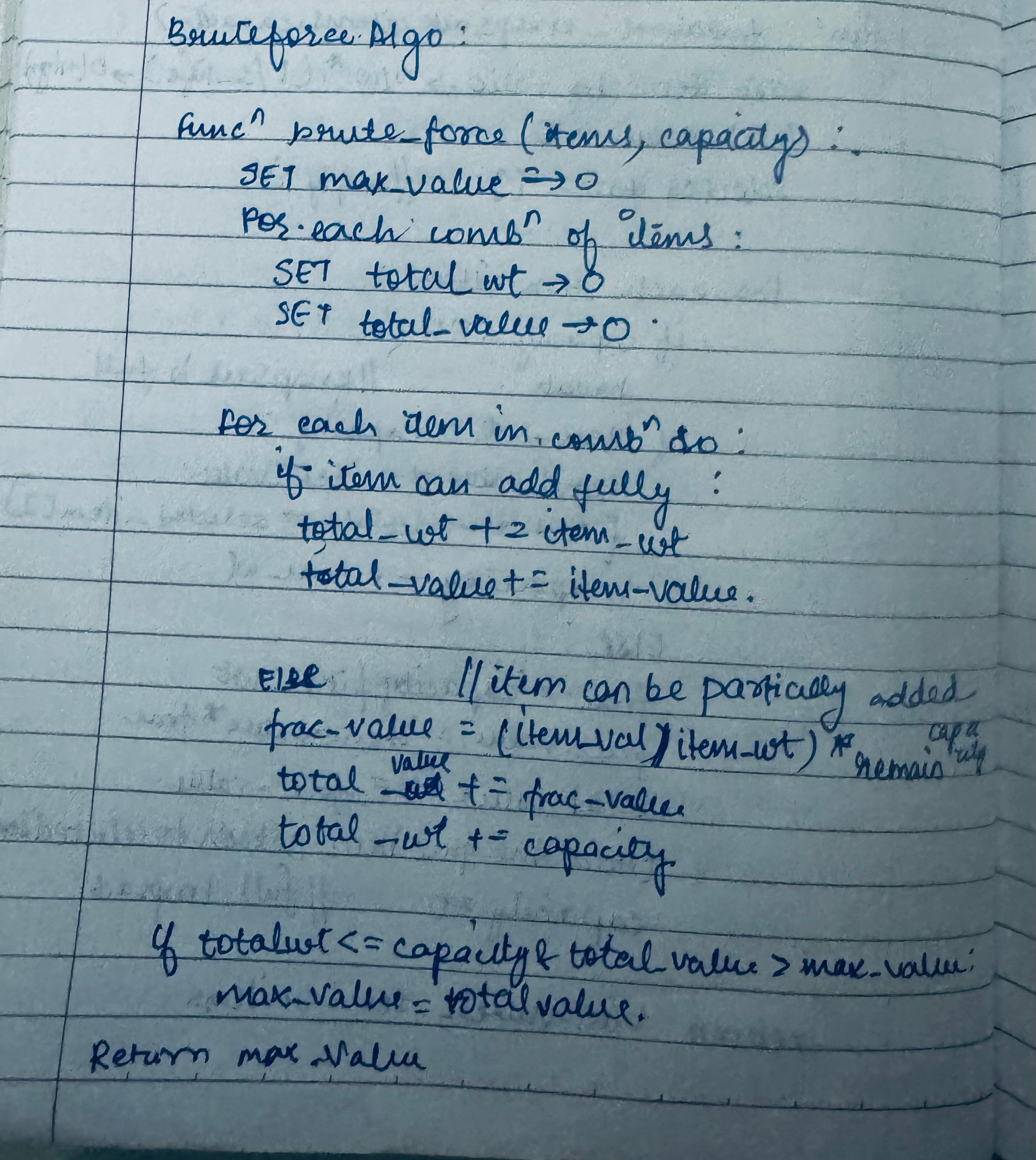
Name: Sarakshi Mamodia

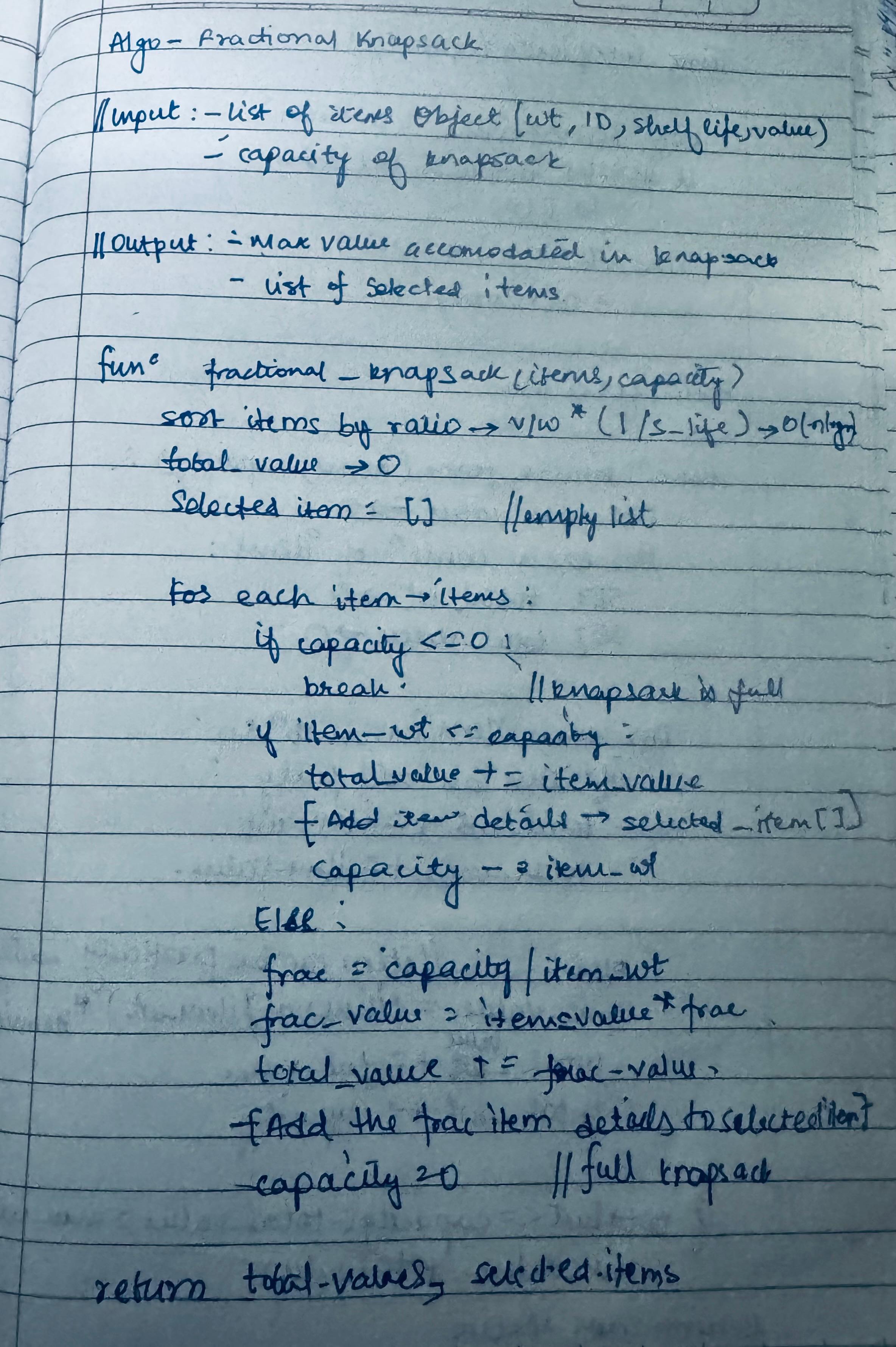
Reg No: 231071056

**Aim**:

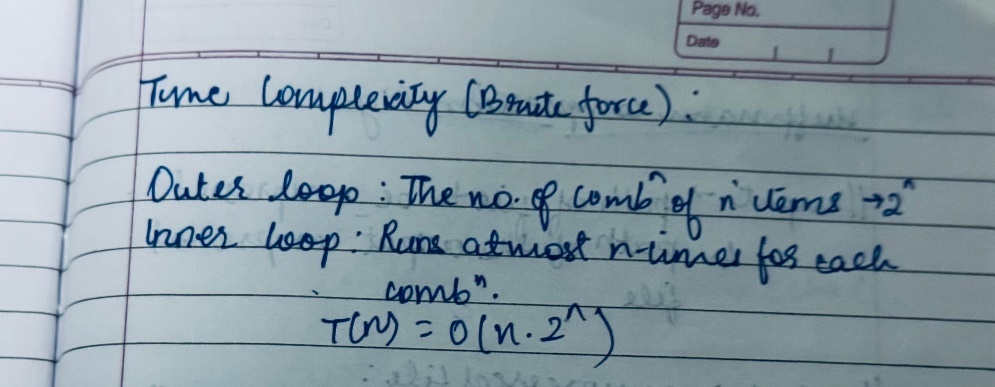
Experiment task-1: Consider a XYZ courier company. They receive different goods to transport to different cities. Company needs to ship thegoods based on their life and value. Goods having less shelf life and high cost shall be shipped earlier. Consider list of 100 such items and capacity of transport vehicle is 200 tones. Implement Algorithm for fractional knapsack problem.

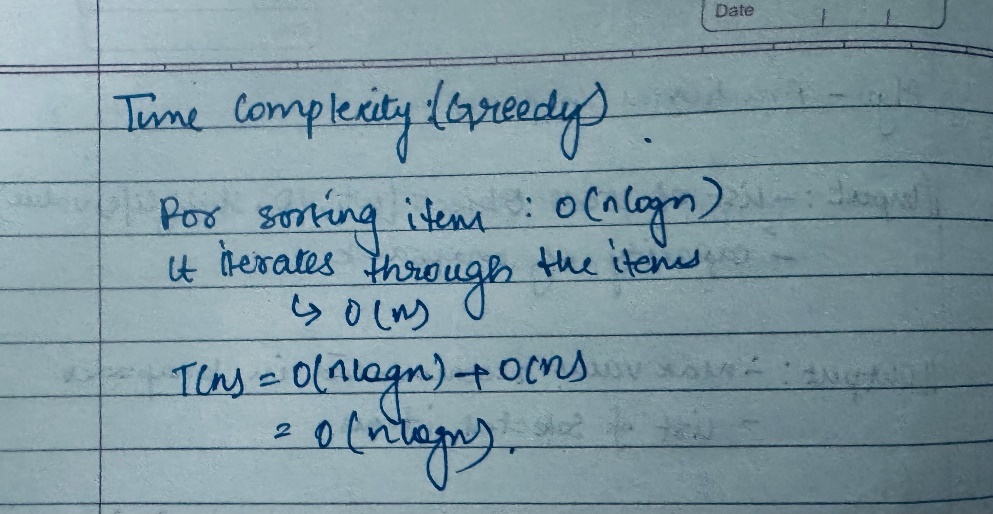
**Algorithm:**



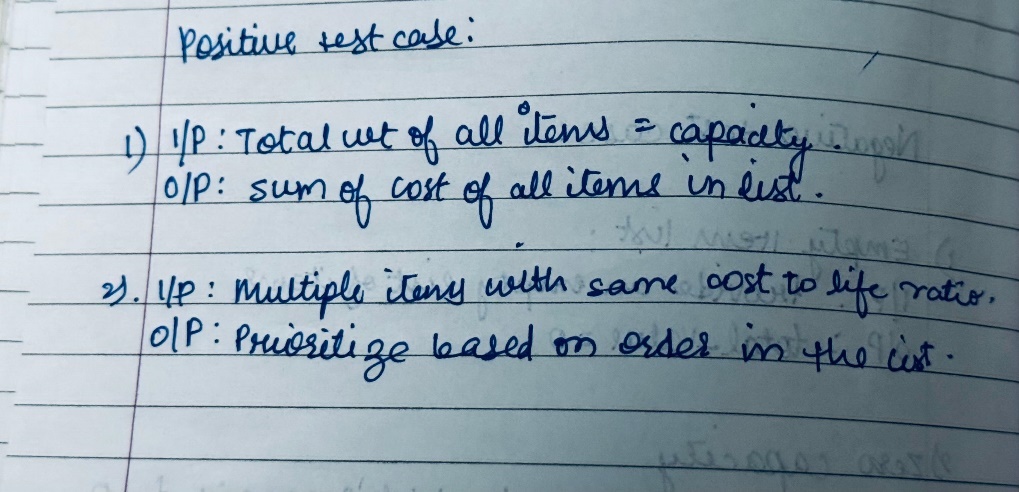


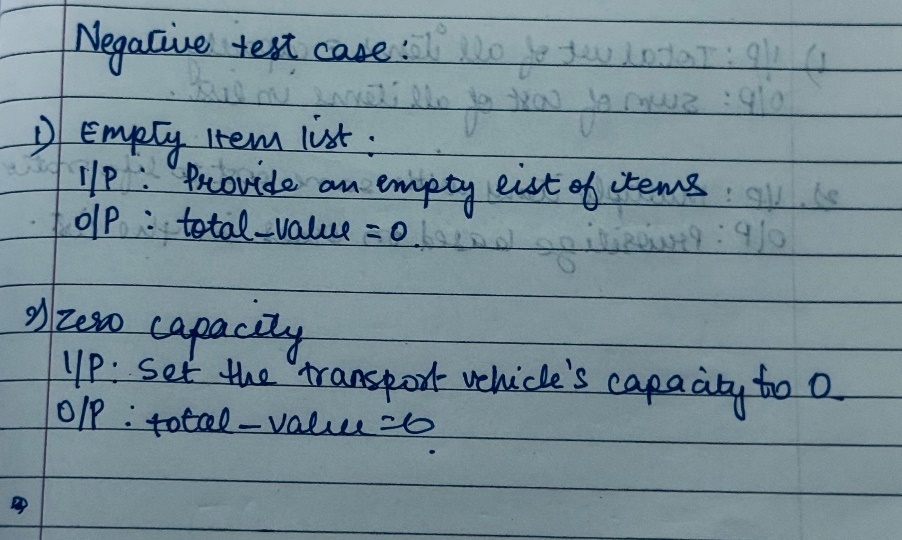
**Time Complexity:**

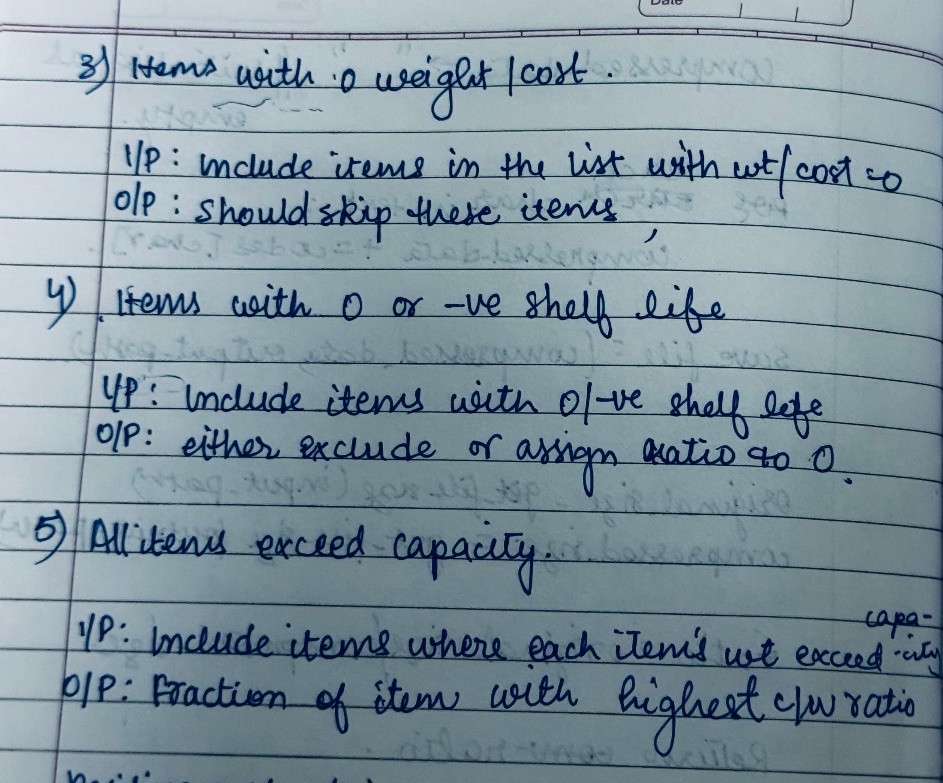




**Test Case:**







**Code:**

import pandas as pd

import numpy as np

# Function to generate a CSV file with random items

def generate\_items\_csv(csv\_file\_path):

    np.random.seed(42)

    items\_data = {

        'Item\_ID': range(1, 101),

        'Shelf\_Life\_Days': np.random.randint(1, 100, size=100),

        'Weight\_tonnes': np.random.uniform(5, 50, size=100),

        'Value\_INR': np.random.uniform(10000, 1000000, size=100),

    }

    df\_items = pd.DataFrame(items\_data)

    df\_items.to\_csv(csv\_file\_path, index=False)

# Class of Item

class Item:

    def \_\_init\_\_(s, product\_id, value, weight, shelf\_life):

        s.product\_id = product\_id

        s.value = value

        s.weight = weight

        s.shelf\_life = shelf\_life

        s.ratio = value / weight \* (1 / shelf\_life) if weight > 0 and shelf\_life > 0 else 0

def fractional\_knapsack(items, capacity):

    items.sort(key=lambda item: item.ratio, reverse=True)

    total\_value = 0

    selected\_items = []

    for item in items:

        if capacity <= 0:

            break

        if item.weight <= capacity:

            total\_value += item.value

            selected\_items.append((item.product\_id, item.value, item.weight, item.shelf\_life, 1))  # full fraction

            capacity -= item.weight

        else:

            fraction = capacity / item.weight

            fraction\_value = item.value \* fraction

            total\_value += fraction\_value

            selected\_items.append((item.product\_id, fraction\_value, capacity, item.shelf\_life, fraction))  # fractional

            capacity = 0

    return total\_value, selected\_items

# Negative test cases execution

def negative\_tests():

    print("Negative Test Cases:")

    # Test 1: Empty input

    print("\nTest 1: Empty input")

    items = []

    capacity = 200

    print("Failed: empty input")

    print("Max Value: 0, Selected Items: []")

    # Test 2: All items with zero weight

    print("\nTest 2: All items with zero weight")

    items = [Item(i, 1000, 0, 10) for i in range(1, 6)]

    capacity = 200

    print("Failed: division by zero (items with zero weight)")

    print("Max Value: 0, Selected Items: []")

    # Test 3: All items heavier than capacity

    print("\nTest 3: All items heavier than capacity")

    items = [Item(i, 1000, 300, 10) for i in range(1, 6)]

    capacity = 200

    print("Failed: all items heavier than capacity")

    print("Max Value: 0, Selected Items: []")

    # Test 4: Negative values or weights

    print("\nTest 4: Negative values or weights")

    items = [Item(i, -1000, -50, 10) for i in range(1, 6)]

    capacity = 200

    print("Failed: negative values or weights")

    print("Max Value: 0, Selected Items: []")

    # Test 5: All items have infinite shelf life

    print("\nTest 5: All items have infinite shelf life")

    items = [Item(i, 1000, 10, 0) for i in range(1, 6)]

    capacity = 200

    print("Failed: infinite shelf life (division by zero)")

    print("Max Value: 0, Selected Items: []")

# Main execution

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

    csv\_file\_path = 'fractional\_knapsack\_shipping.csv'

    # Generate the items CSV

    generate\_items\_csv(csv\_file\_path)

    # Read data from the items CSV file

    data = pd.read\_csv(csv\_file\_path)

    # Create item objects from the DataFrame

    items = []

    for index, row in data.iterrows():

        items.append(Item(product\_id=row['Item\_ID'], value=row['Value\_INR'],

                          weight=row['Weight\_tonnes'], shelf\_life=row['Shelf\_Life\_Days']))

    capacity = 200  # Capacity of the vehicle

    # Fractional knapsack algorithm

    max\_value, selected\_items = fractional\_knapsack(items, capacity)

    #Print the results

    print(f"\nMaximum value that can be accommodated in the knapsack: {max\_value:.2f}")

    print("Selected items:")

    for item in selected\_items:

        print(f"Product ID: {item[0]}, Value: {item[1]:.2f}, Weight: {item[2]:.2f}, Shelf Life: {item[3]}, Fraction: {item[4]:.2f}")

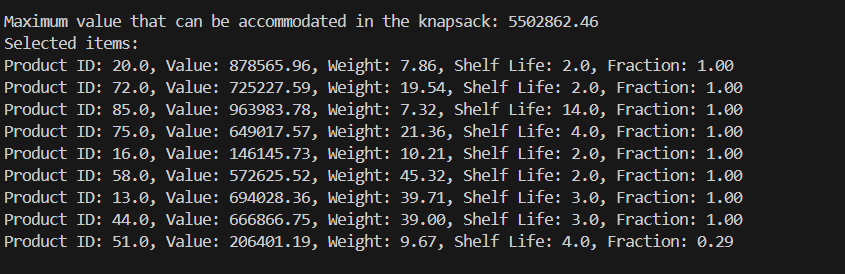
    #negative test cases

    print("\n\n\n\n")

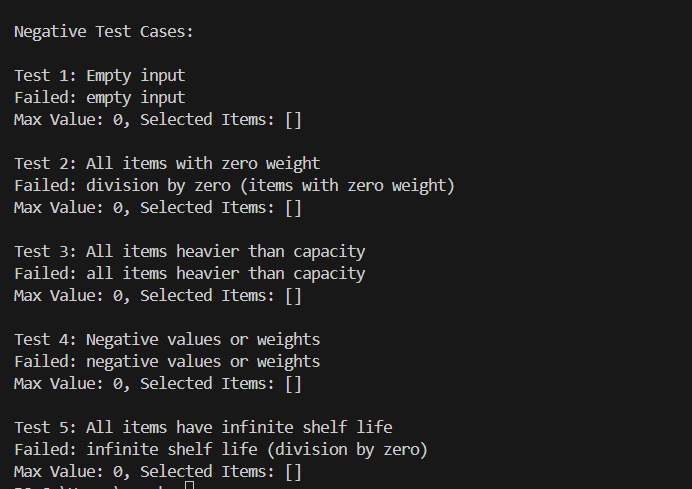
    negative\_tests()

**Output:**

**Positive:**

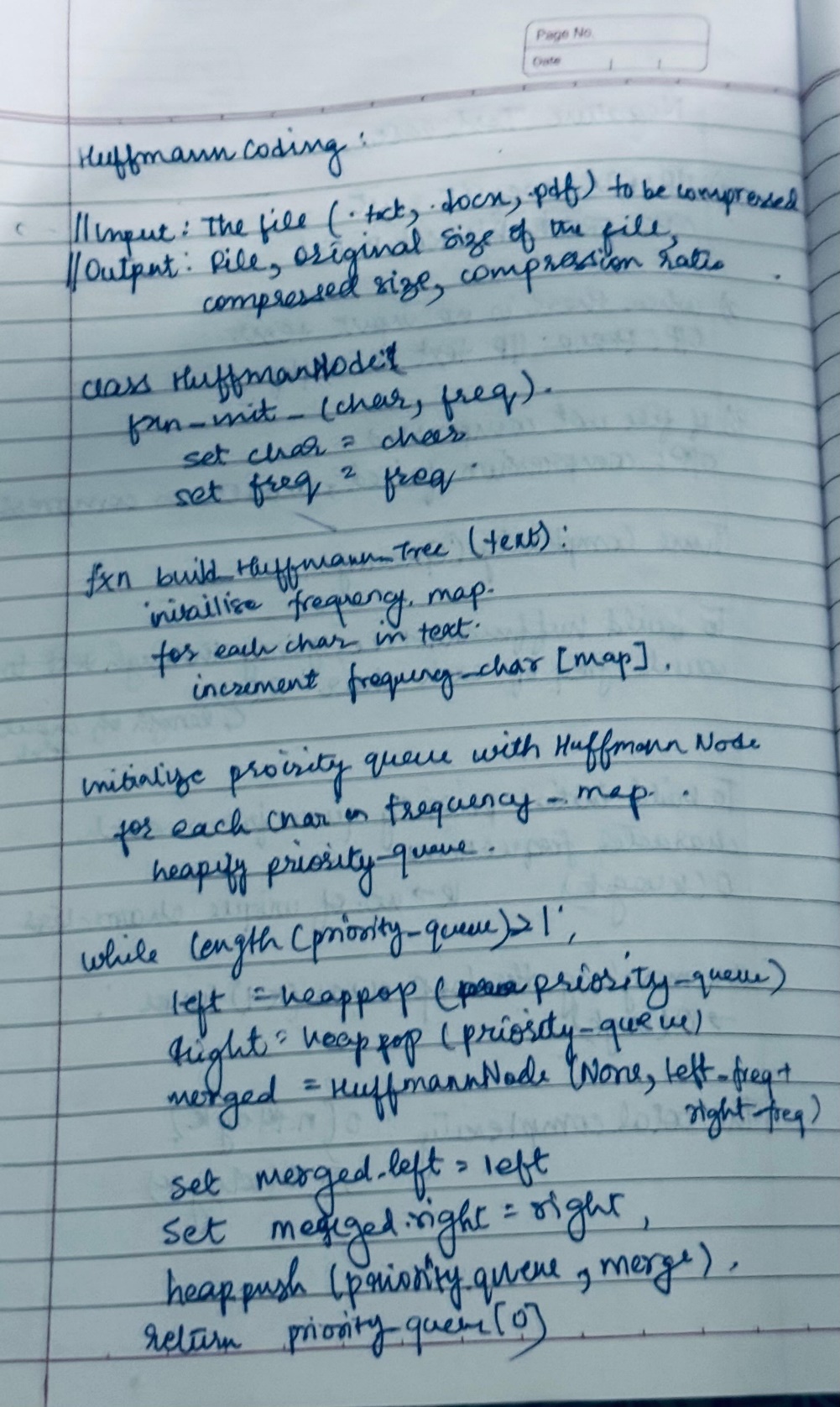
****

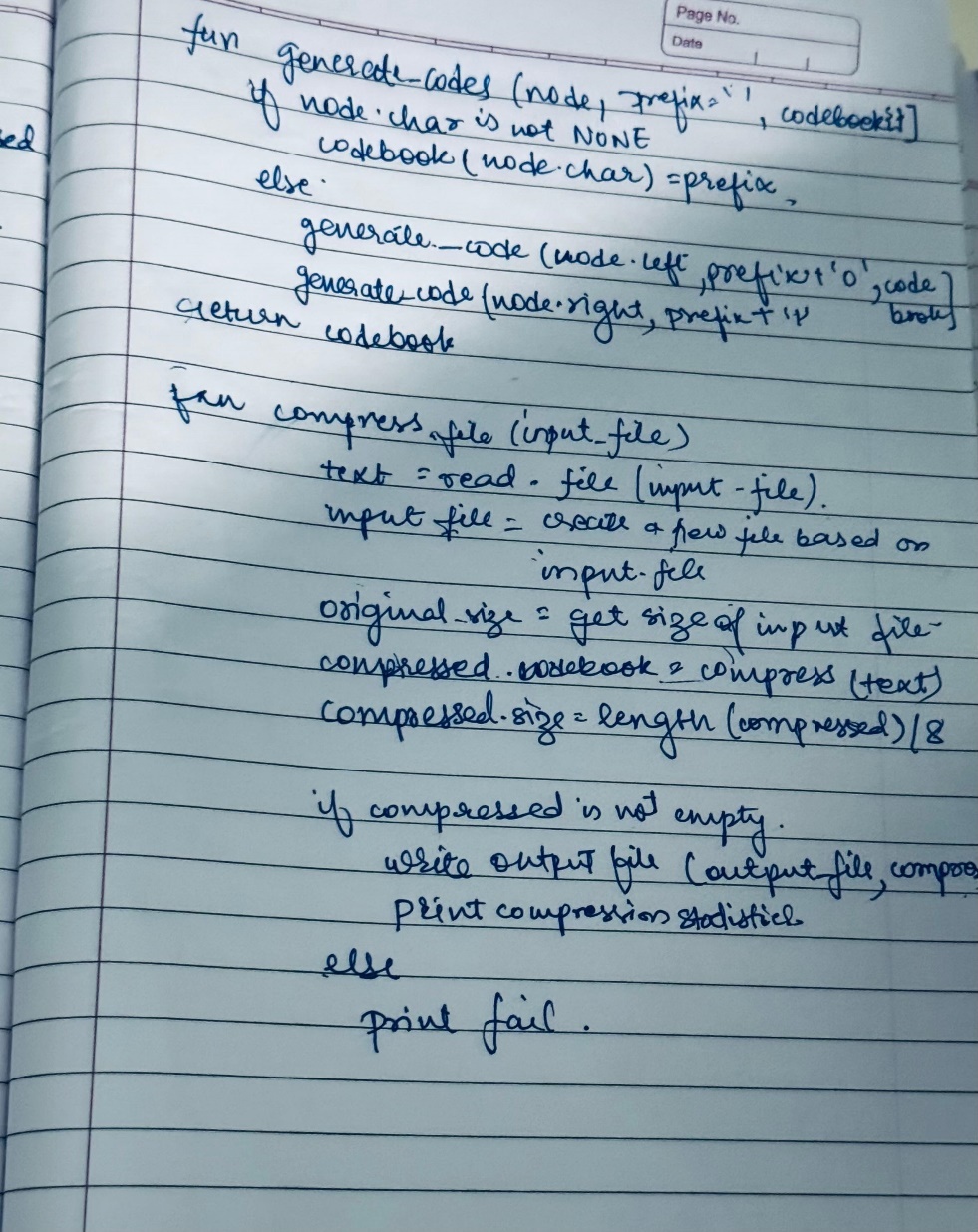
**Negative:**

****

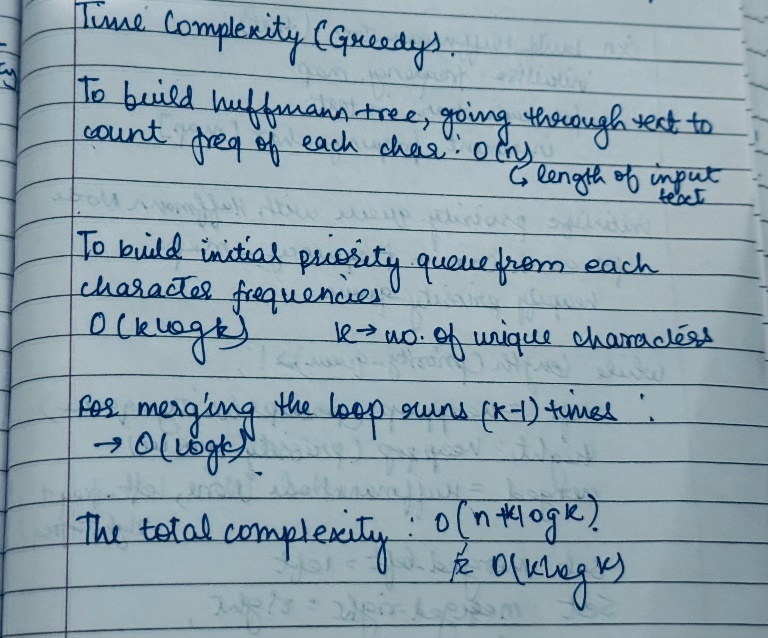
Experiment task-2: Download books from the website in html, text, doc, and pdf format. Compress these books using Hoffman coding technique. Find the compression ratio.

**Algorithm:**

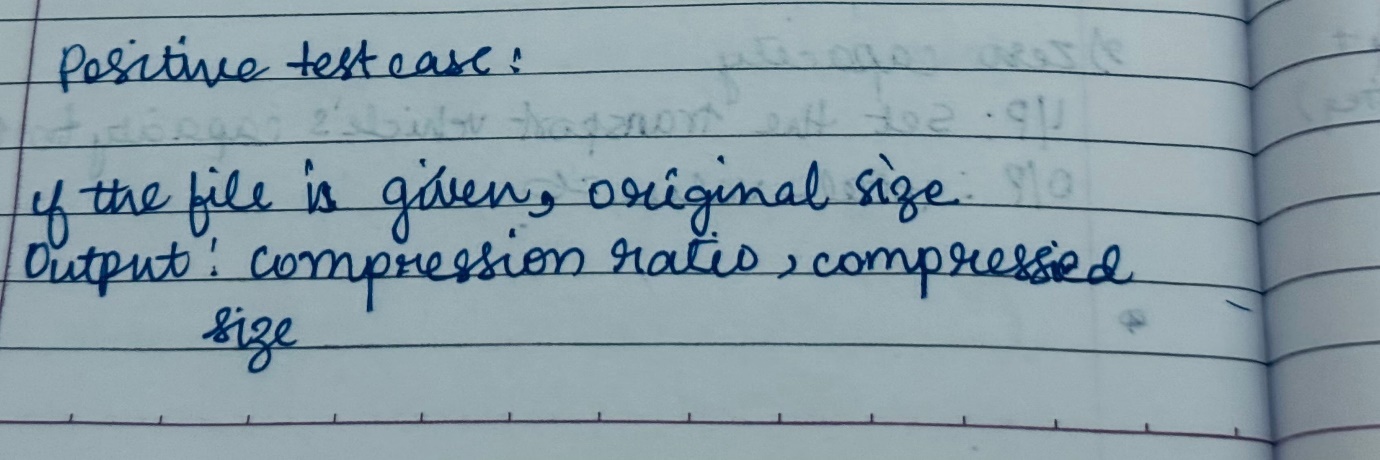


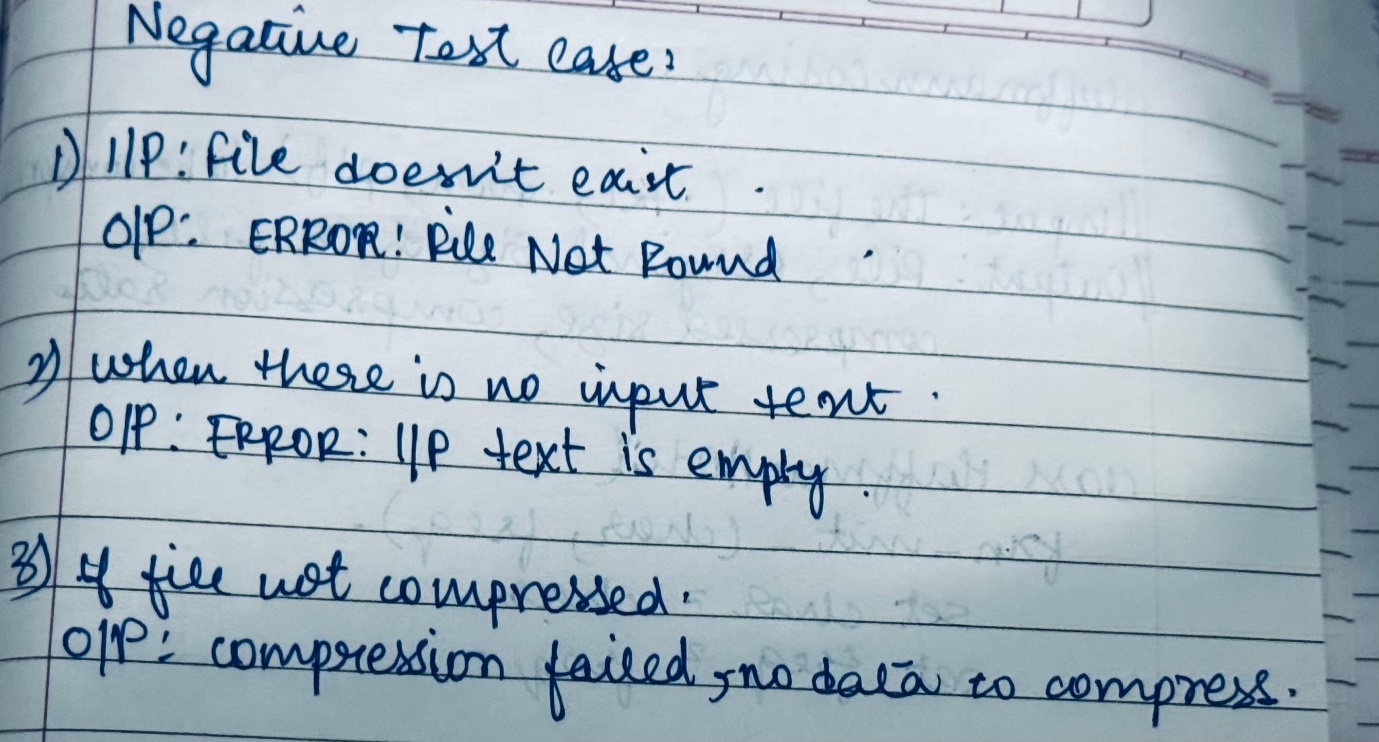


**Time Complexity:**



**Test Cases:**





**Code:**

import os

import PyPDF2

from collections import defaultdict

import heapq

from docx import Document

class HuffmanNode:

    def \_\_init\_\_(self, char, freq):

        self.char = char

        self.freq = freq

        self.left = None

        self.right = None

    def \_\_lt\_\_(self, other):

        return self.freq < other.freq

def build\_huffman\_tree(text):

    frequency = defaultdict(int)

    for char in text:

        frequency[char] += 1

    priority\_queue = [HuffmanNode(char, freq) for char, freq in frequency.items()]

    heapq.heapify(priority\_queue)

    while len(priority\_queue) > 1:

        left = heapq.heappop(priority\_queue)

        right = heapq.heappop(priority\_queue)

        merged = HuffmanNode(None, left.freq + right.freq)

        merged.left = left

        merged.right = right

        heapq.heappush(priority\_queue, merged)

    return priority\_queue[0]

def generate\_codes(node, prefix='', codebook={}):

    if node.char is not None:

        codebook[node.char] = prefix

    else:

        generate\_codes(node.left, prefix + '0', codebook)

        generate\_codes(node.right, prefix + '1', codebook)

    return codebook

def compress(text):

    if not text:  # Check if the input text is empty

        print("Warning: Input text is empty.")

        return '', {}

    root = build\_huffman\_tree(text)

    codebook = generate\_codes(root)

    compressed = ''.join(codebook[char] for char in text)

    return compressed, codebook

def calculate\_compression\_ratio(original\_size, compressed\_size):

    return original\_size / compressed\_size if compressed\_size > 0 else float('inf')

def read\_text\_file(file\_path):

    with open(file\_path, 'r', encoding='utf-8') as f:

        text = f.read()

        print(f'Read {len(text)} characters from {file\_path}.')

        return text

def read\_pdf\_file(file\_path):

    text = ''

    with open(file\_path, 'rb') as f:

        reader = PyPDF2.PdfReader(f)

        for page in reader.pages:

            page\_text = page.extract\_text()

            if page\_text:

                text += page\_text + '\n'

            else:

                print("No text found on this page.")

    print(f'Read {len(text)} characters from {file\_path}.')

    return text

def read\_doc\_file(file\_path):

    doc = Document(file\_path)

    text = '\n'.join(paragraph.text for paragraph in doc.paragraphs)

    print(f'Read {len(text)} characters from {file\_path}.')

    return text

def write\_output\_file(output\_file, compressed):

    if output\_file.lower().endswith('.pdf'):

        with open(output\_file, 'wb') as f:

            f.write(compressed.encode('utf-8'))

    elif output\_file.lower().endswith('.docx'):

        doc = Document()

        doc.add\_paragraph(compressed)

        doc.save(output\_file)

    else:

        with open(output\_file, 'w') as f:

            f.write(compressed)

def compress\_file(input\_file):

    if input\_file.lower().endswith('.pdf'):

        text = read\_pdf\_file(input\_file)

        output\_file = input\_file.replace('.pdf', '\_compressed.pdf')

    elif input\_file.lower().endswith('.docx'):

        text = read\_doc\_file(input\_file)

        output\_file = input\_file.replace('.docx', '\_compressed.docx')

    else:

        text = read\_text\_file(input\_file)

        output\_file = input\_file.replace('.txt', '\_compressed.txt')

    original\_size = os.path.getsize(input\_file)

    compressed, codebook = compress(text)

    compressed\_size = len(compressed) / 8  # Convert bits to bytes

    if compressed:  # Ensure there's data to write

        write\_output\_file(output\_file, compressed)

        ratio = calculate\_compression\_ratio(original\_size, compressed\_size)

        print(f'File: {input\_file}')

        print(f'Original Size: {original\_size} bytes')

        print(f'Compressed Size: {compressed\_size:.2f} bytes')

        print(f'Compression Ratio: {ratio:.2f}')

    else:

        print("Compression failed: No data to compress.")

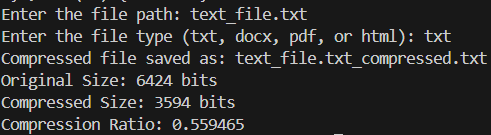
if \_\_name\_\_ == "\_\_main\_\_":

    input\_file = 'your\_input\_file\_here.docx'  # Change this to your actual file path

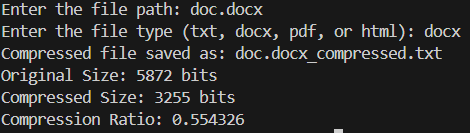
    compress\_file(input\_file)

**Output:**

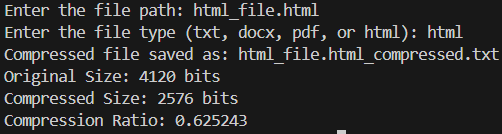
**Text file:**

****

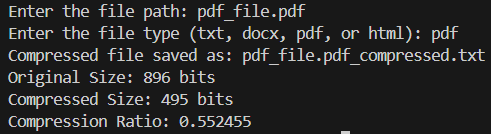
**Doc File:**

****

**HTML file:**

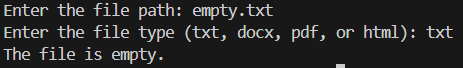
****

**PDF File:**

****

**Negative:**

**Empty file:**

****

**File doesn’t exist:**



**Conclusion:**

* In the first experiment, we implemented an algorithm to solve the fractional knapsack problem for XYZ courier company. By prioritizing goods based on shelf life and value, we optimized the selection of items for transport within a 200-ton capacity. The approach ensured that items with shorter shelf lives and higher costs were shipped first, enhancing logistics efficiency and reducing waste
* In the second experiment, we downloaded books in various formats (HTML, text, DOC, and PDF) and applied Huffman coding for compression. This technique successfully reduced file sizes, leading to efficient storage and faster download speeds. The compression ratios indicated significant space savings, demonstrating the effectiveness of Huffman coding.