Name: Divyansh Dubey

Roll no: 231070001

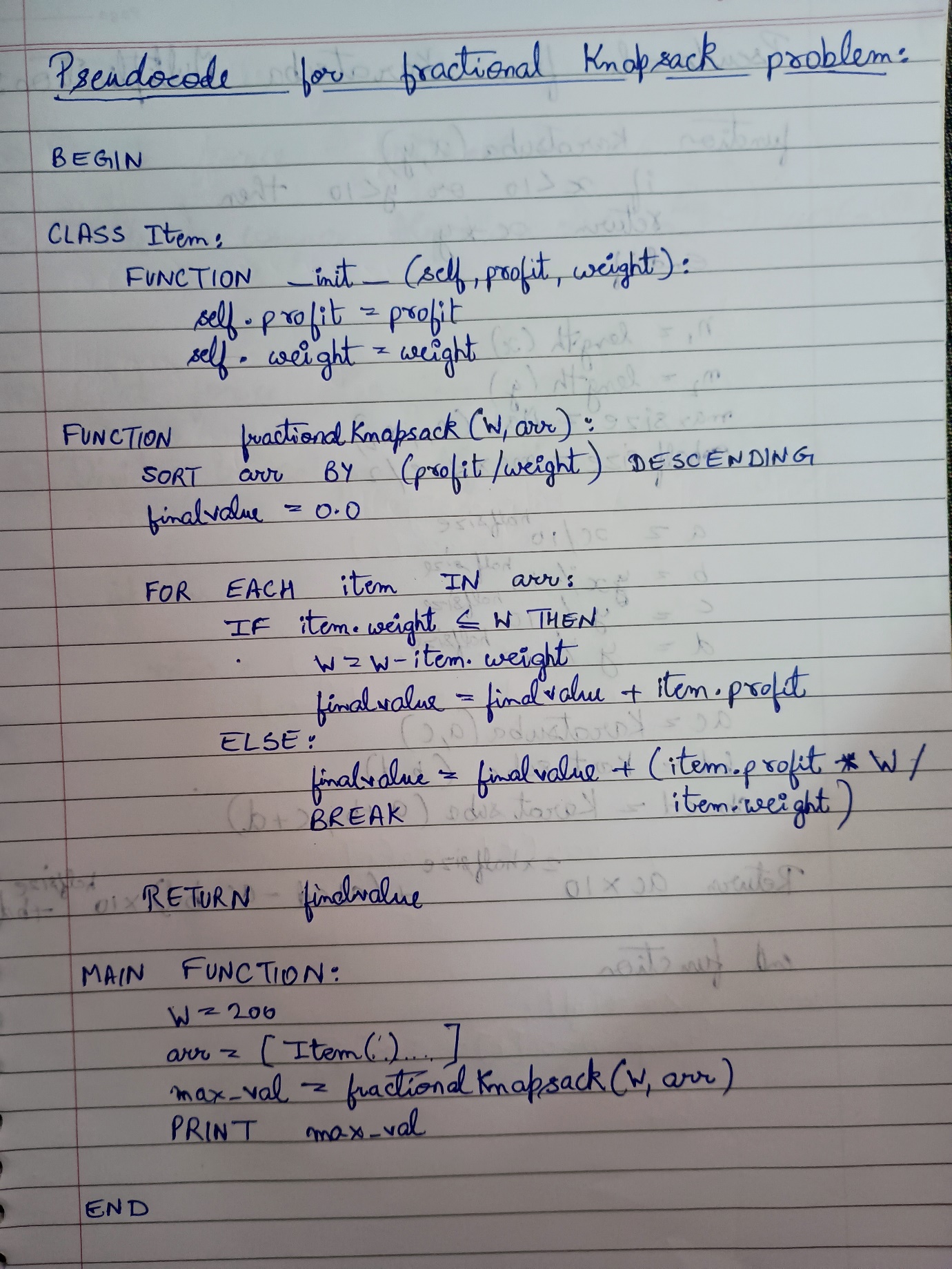
Course: B. Tech Computer Engineering

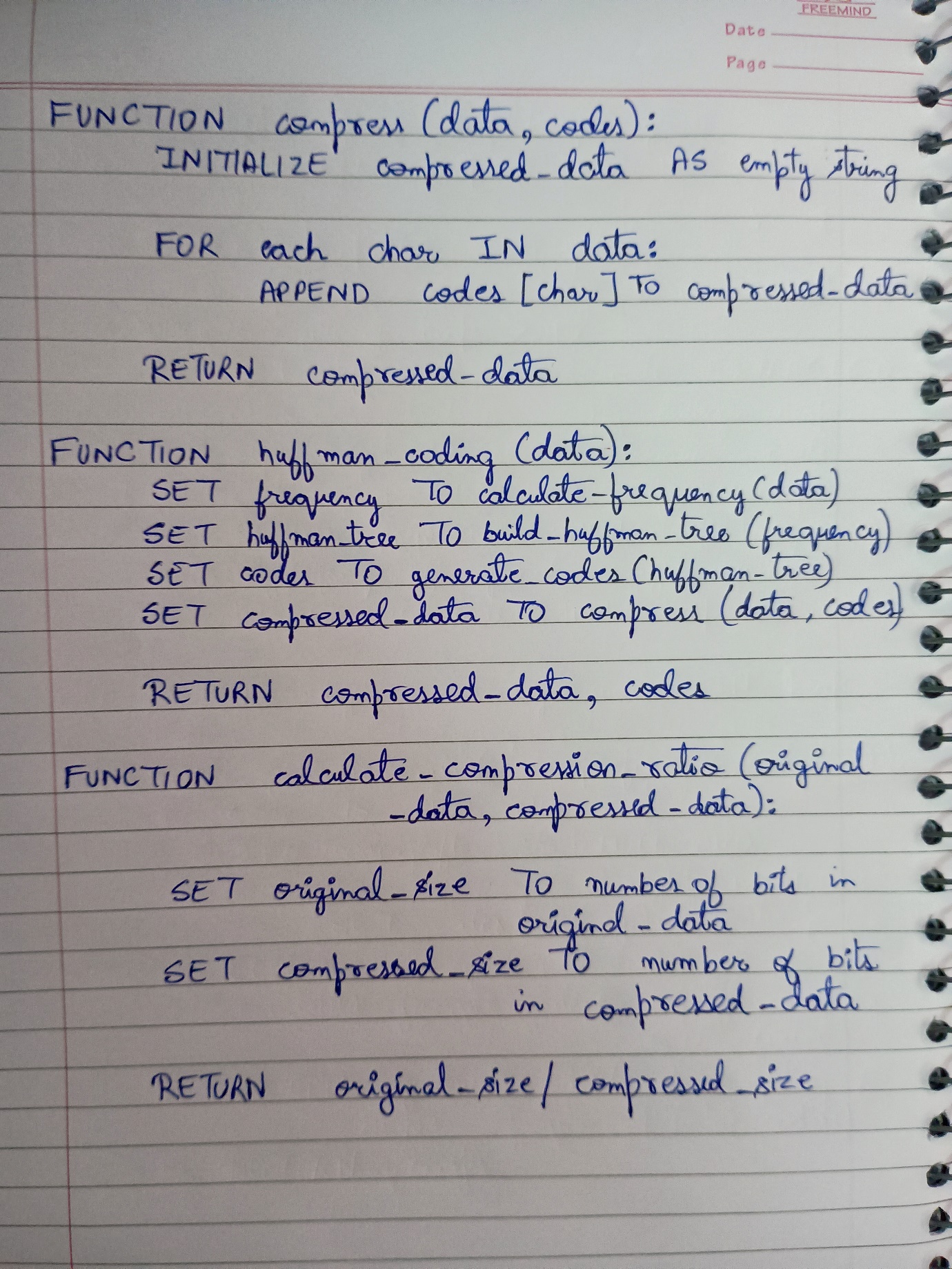
Batch: A

Experiment- 5

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 vehical is 200 tones. Implement Algorithm for fractional knapsack problem.  
  
  
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.

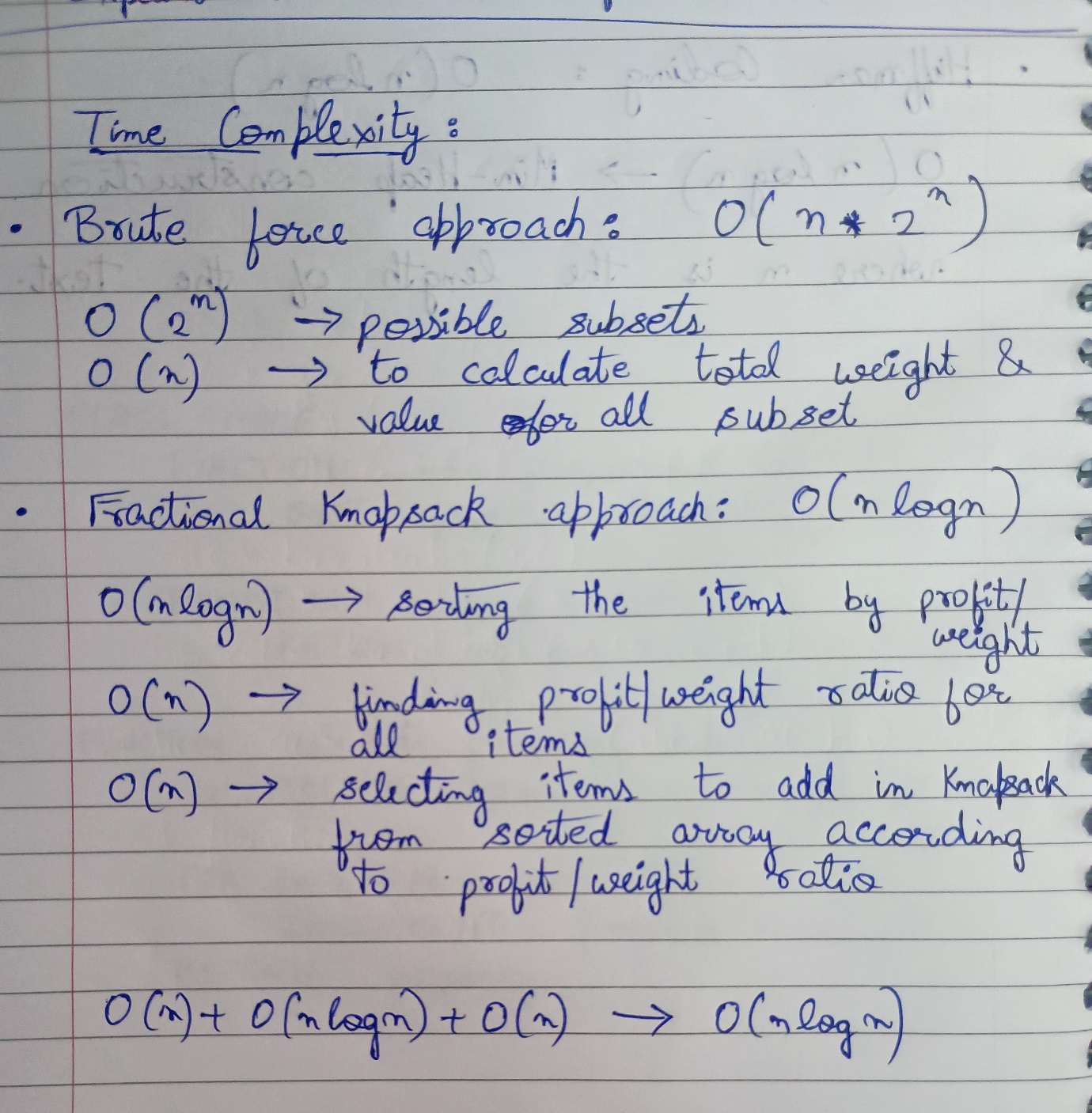
Algorithm:



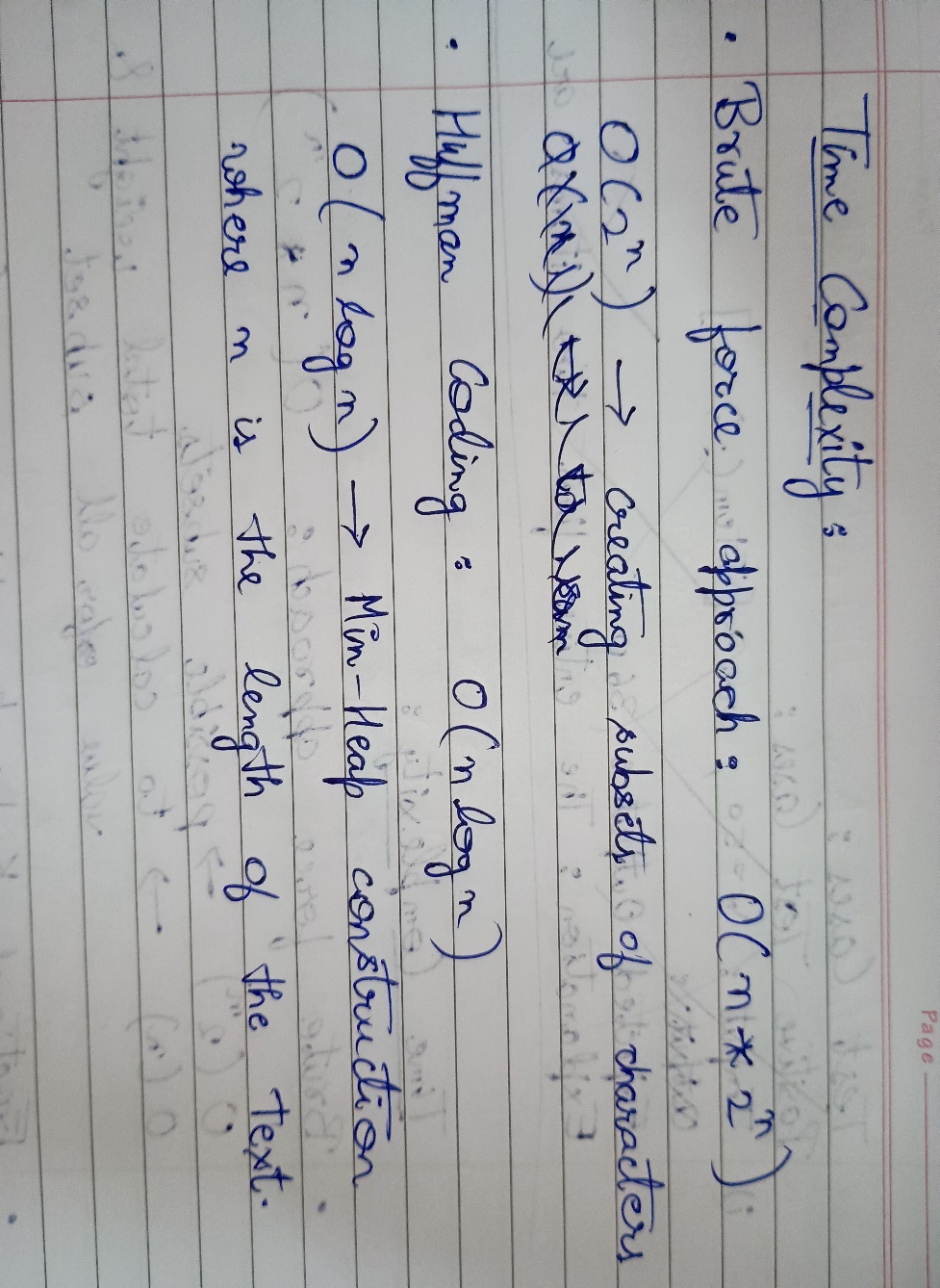


Time Complexity:

For Experiment 1:



For Experiment 2:



Code:

Experiment 1:

# Structure for an item which stores weight and

# corresponding value of Item

class Item:

  def \_\_init\_\_(self, profit, weight):

    self.profit = profit

    self.weight = weight

# Main greedy function to solve problem

def fractionalKnapsack(W, arr):

  # Sorting Item on basis of ratio

  arr.sort(key=lambda x: (x.profit/x.weight), reverse=True)

  # Result(value in Knapsack)

  final\_value = 0.0

  # Looping through all Items

  for item in arr:

    # If adding Item won't overflow,

    # add it completely

    if item.weight <= W:

      W -= item.weight

      final\_value += item.profit

    # If we can't add current Item,

    # add fractional part of it

    else:

      final\_value += item.profit \* W / item.weight

      break

  # Returning final value

  return final\_value

W = 50

arr = [  Item(60, 10), Item(10, 20), Item(12, 30), Item(90, 20), Item(20, 50),

        Item(15, 40), Item(18, 30), Item(30, 60), Item(35, 70), Item(40, 80),

        Item(25, 30), Item(5, 90), Item(50, 10), Item(60, 10), Item(70, 10),

        Item(80, 10), Item(10, 10), Item(10, 10), Item(10, 10), Item(10, 70),

        Item(13, 10), Item(14, 10), Item(10, 20), Item(10, 20), Item(10, 20),

        Item(18, 20), Item(19, 20), Item(20, 20), Item(20, 20), Item(20, 20),

        Item(23, 20), Item(24, 20), Item(20, 30), Item(20, 30), Item(20, 30),

        Item(28, 30), Item(29, 30), Item(30, 30), Item(30, 30), Item(32, 30),

        Item(33, 30), Item(30, 30), Item(30, 40), Item(30, 40), Item(30, 40),

        Item(38, 40), Item(30, 40), Item(40, 40), Item(40, 40), Item(42, 40),

        Item(43, 40), Item(44, 40), Item(40, 50), Item(40, 50), Item(47, 50),

        Item(48, 50), Item(40, 50), Item(50, 50), Item(51, 50), Item(50, 50),

        Item(50, 50), Item(50, 59), Item(50, 60), Item(70, 60), Item(50, 60),

        Item(50, 60), Item(50, 60), Item(60, 10), Item(60, 60), Item(60, 60),

        Item(60, 60), Item(60, 69), Item(60, 70), Item(10, 70), Item(60, 70),

        Item(60, 70), Item(90, 70), Item(70, 70), Item(70, 70), Item(72, 70),

        Item(70, 70), Item(70, 70), Item(70, 80), Item(76, 80), Item(70, 20),

        Item(70, 80), Item(70, 84), Item(80, 80), Item(80, 80), Item(80, 80),

        Item(80, 80), Item(80, 80), Item(80, 90), Item(80, 90), Item(80, 90),

        Item(80, 90), Item(80, 90), Item(90, 90), Item(90, 90), Item(90, 90),

        Item(90, 90), Item(94, 90), Item(90, 10) ]

  # Function call

max\_val = fractionalKnapsack(W, arr)

print(max\_val)

Experiment 2:

import PyPDF2

import heapq

from collections import defaultdict

# Function to extract text from PDF

def extract\_text\_from\_pdf(path\_pdf):

    pdf\_file = open(path\_pdf, 'rb')

    pdf\_reader = PyPDF2.PdfReader(path\_pdf)

    text = ""

    for page\_num in range(len(pdf\_reader.pages)):

        page = pdf\_reader.pages[page\_num]

        text += page.extract\_text()

    pdf\_file.close()

    return text

# Huffman Coding implementation

class Node:

    def \_\_init\_\_(self, freq, symbol, left=None, right=None):

        self.freq = freq

        self.symbol = symbol

        self.left = left

        self.right = right

        self.huff = ''

    def \_\_lt\_\_(self, nxt):

        return self.freq < nxt.freq

def calculate\_frequency(data):

    frequency = defaultdict(int)

    for char in data:

        frequency[char] += 1

    return frequency

def build\_huffman\_tree(frequency):

    heap = [Node(freq, symbol) for symbol, freq in frequency.items()]

    heapq.heapify(heap)

    while len(heap) > 1:

        left = heapq.heappop(heap)

        right = heapq.heappop(heap)

        newNode = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)

        heapq.heappush(heap, newNode)

    return heap[0]

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

    if node is not None:

        if not node.left and not node.right:

            codes[node.symbol] = prefix

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

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

    return codes

def compress(data, codes):

    compressed\_data = ''.join([codes[char] for char in data])

    return compressed\_data

def huffman\_coding(data):

    frequency = calculate\_frequency(data)

    huffman\_tree = build\_huffman\_tree(frequency)

    codes = generate\_codes(huffman\_tree)

    compressed\_data = compress(data, codes)

    return compressed\_data, codes

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

    original\_size = len(original\_data) \* 8

    compressed\_size = len(compressed\_data)

    return original\_size / compressed\_size

# Extract text from PDF

pdf\_path = "The-Richest-Man-In-Babylon.pdf"

data = extract\_text\_from\_pdf(pdf\_path)

# Apply Huffman coding

compressed\_data, codes = huffman\_coding(data)

compression\_ratio = calculate\_compression\_ratio(data, compressed\_data)

print(f"Compression Ratio: {compression\_ratio}")

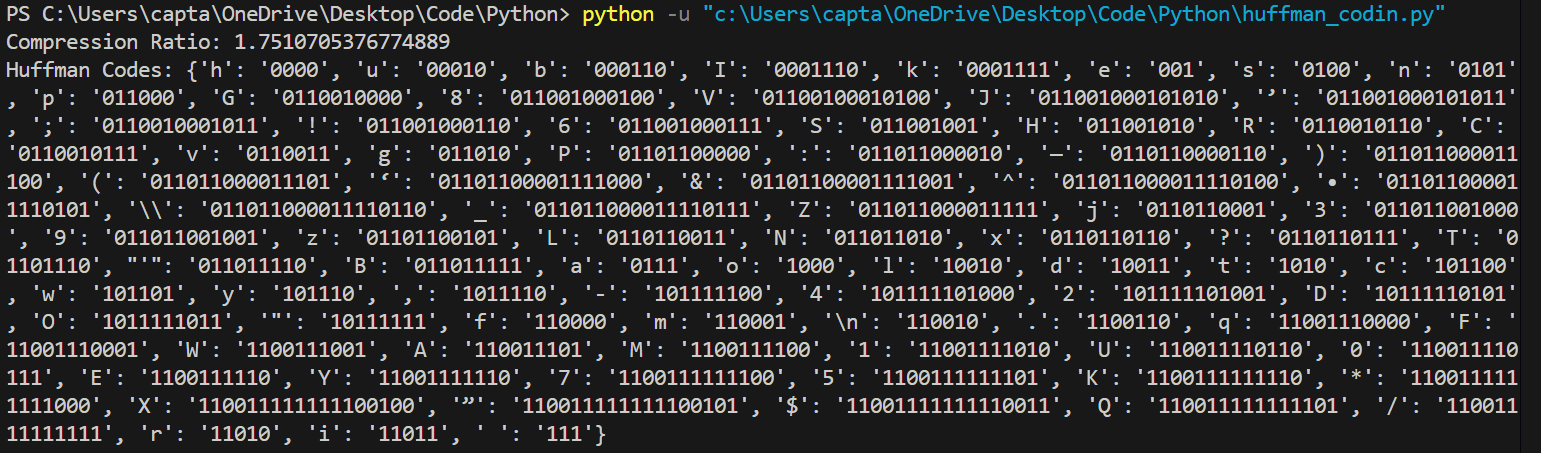
print(f"Huffman Codes: {codes}")

Output:

Experiment 1:



Experiment 2:



Conclusion:

Debugging tools helps a lot programmers to find out various bugs and fixes it efficiently.This shows that fractional knapsack problem is a very powerful and optimized algorithm to choose between items giving maximum profit with least cost with a worst time complexity of O(n\*log n). Also, Huffman coding helps in encoding and decoding the text into binary form in an space efficient way with a time complexity of O(n\*log n).